EXHIBIT A

Introduction

This part presents several algorithms that solve the following ***sorting problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as a linked list.

The structure of the data

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a ***record***. Each record contains a ***key***, which is the value to be sorted, and the remainder of the record consists of ***satellite data***, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

In a sense, it is these implementation details that distinguish an algorithm from a full-blown program. Whether we sort individual numbers or large records that contain numbers is irrelevant to the *method* by which a sorting procedure determines the sorted order. Thus, when focusing on the problem of sorting, we typically assume that the input consists only of numbers. The translation of an algorithm for sorting numbers into a program for sorting records is conceptually straightforward, although in a given engineering situation there may be other subtleties that make the actual programming task a challenge.

Sorting algorithms

We introduced two algorithms that sort n real numbers in Chapter 1. Insertion sort takes $\Theta(n^2)$ time in the worst case. Because its inner loops are tight, however, it is a fast in-place sorting algorithm for small input sizes. (Recall that a sorting algorithm sorts ***in place*** if only a constant number of elements of the input array are ever stored outside the array.) Merge sort has a better asymptotic running time, $\Theta(n \lg n)$, but the MERGE procedure it uses does not operate in place.

In this part, we shall introduce two more algorithms that sort arbitrary real numbers. Heapsort, presented in Chapter 7, sorts n numbers in place in $O(n \lg n)$ time. It uses an important data structure, called a heap, to implement a priority queue.

Quicksort, in Chapter 8, also sorts n numbers in place, but its worst-case running time is $\Theta(n^2)$. Its average-case running time is $\Theta(n \lg n)$, though, and it generally outperforms heapsort in practice. Like insertion sort, quicksort has tight code, so the hidden constant factor in its running time is small. It is a popular algorithm for sorting large input arrays.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements. Chapter 9 begins by introducing the decision-tree model in order to study the performance limitations of comparison sorts. Using this model, we prove a lower bound of $\Omega(n \lg n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

Chapter 9 then goes on to show that we can beat this lower bound of $\Omega(n \lg n)$ if we can gather information about the sorted order of the input by means other than comparing elements. The counting sort algorithm, for example, assumes that the input numbers are in the set $\{1, 2, \ldots, k\}$. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $O(k + n)$ time. Thus, when $k = O(n)$, counting sort runs in time that is linear in the size of the input array. A related algorithm, radix sort, can be used to extend the range of counting sort. If there are n integers to sort, each integer has d digits, and each digit is in the set $\{1, 2, \ldots, k\}$, radix sort can sort the numbers in $O(d(n + k))$ time. When d is a constant and k is $O(n)$, radix sort runs in linear time. A third algorithm, bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array. It can sort n real numbers uniformly distributed in the half-open interval $[0, 1)$ in average-case $O(n)$ time.

Order statistics

The ith order statistic of a set of n numbers is the ith smallest number in the set. One can, of course, select the ith order statistic by sorting the input and indexing the ith element of the output. With no assumptions about the input distribution, this method runs in $\Omega(n \lg n)$ time, as the lower bound proved in Chapter 9 shows.

In Chapter 10, we show that we can find the ith smallest element in $O(n)$ time, even when the elements are arbitrary real numbers. We present an algorithm with tight pseudocode that runs in $O(n^2)$ time in the worst case, but linear time on average. We also give a more complicated algorithm that runs in $O(n)$ worst-case time.

Background

Although most of this part does not rely on difficult mathematics, some sections do require mathematical sophistication. In particular, the average-case analyses of quicksort, bucket sort, and the order-statistic algorithm use probability, which is reviewed in Chapter 6. The analysis of the worst-case linear-time algorithm for the order statistic involves somewhat more sophisticated mathematics than the other worst-case analyses in this part.

7 Heapsort

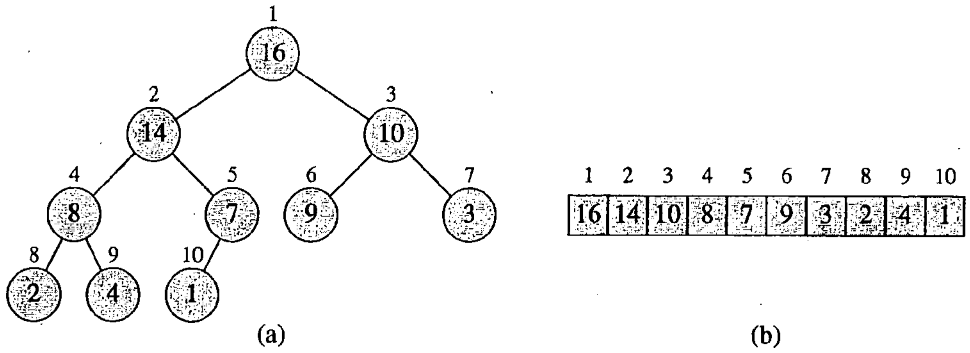
In this chapter, we introduce another sorting algorithm. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: the use of a data structure, in this case one we call a "heap," to manage information during the execution of the algorithm. Not only is the heap data structure useful for heapsort, it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

We note that the term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming language Lisp provides. Our heap data structure is *not* garbage-collected storage, and whenever we refer to heaps in this book, we shall mean the structure defined in this chapter.

7.1 Heaps

The ***(binary) heap*** data structure is an array object that can be viewed as a complete binary tree (see Section 5.5.3), as shown in Figure 7.1. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: *length*$[A]$, which is the number of elements in the array, and *heap-size*$[A]$, the number of elements in the heap stored within array A. That is, although $A[1\,..\,\mathit{length}[A]]$ may contain valid numbers, no element past $A[\mathit{heap\text{-}size}[A]]$, where $\mathit{heap\text{-}size}[A] \leq \mathit{length}[A]$, is an element of the heap. The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent PARENT(i), left child LEFT(i), and right child RIGHT(i) can be computed simply:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Figure 7.1 A heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

```
PARENT($i$)
    return $\lfloor i/2 \rfloor$

LEFT($i$)
    return $2i$

RIGHT($i$)
    return $2i + 1$
```

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left one bit position and shifting in a 1 as the low-order bit. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. In a good implementation of heapsort, these three procedures are often implemented as "macros" or "in-line" procedures.

Heaps also satisfy the ***heap property***: for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i] , \tag{7.1}$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a heap is stored at the root, and the subtrees rooted at a node contain smaller values than does the node itself.

We define the ***height*** of a node in a tree to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the tree to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 7.1-2). We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents five basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the heap property (7.1).
- The BUILD-HEAP procedure, which runs in linear time, produces a heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The EXTRACT-MAX and INSERT procedures, which run in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

Exercises

7.1-1
What are the minimum and maximum numbers of elements in a heap of height h?

7.1-2
Show that an n-element heap has height $\lfloor \lg n \rfloor$.

7.1-3
Show that the largest element in a subtree of a heap is at the root of the subtree.

7.1-4
Where in a heap might the smallest element reside?

7.1-5
Is an array that is in reverse sorted order a heap?

7.1-6
Is the sequence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a heap?

7.2 Maintaining the heap property

HEAPIFY is an important subroutine for manipulating heaps. Its inputs are an array A and an index i into the array. When HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but that $A[i]$ may be smaller than its children, thus violating the heap property (7.1). The function of HEAPIFY is to let the value at $A[i]$ "float down" in the heap so that the subtree rooted at index i becomes a heap.

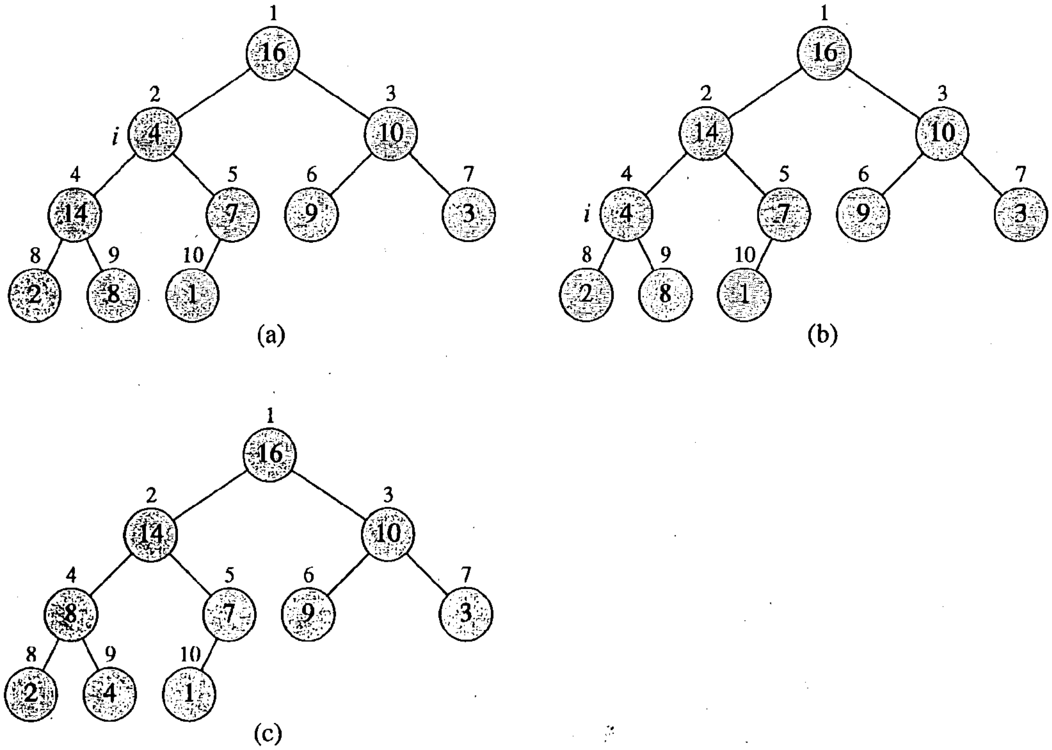


Figure 7.2 The action of HEAPIFY(A, 2), where *heap-size*[A] = 10. (**a**) The initial configuration of the heap, with $A[2]$ at node $i = 2$ violating the heap property since it is not larger than both children. The heap property is restored for node 2 in (**b**) by exchanging $A[2]$ with $A[4]$, which destroys the heap property for node 4. The recursive call HEAPIFY(A, 4) now sets $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (**c**), node 4 is fixed up, and the recursive call HEAPIFY(A, 9) yields no further change to the data structure.

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  if l ≤ heap-size[A] and A[l] > A[i]
     then largest ← l
     else largest ← i
  if r ≤ heap-size[A] and A[r] > A[largest]
     then largest ← r
  if largest ≠ i
     then exchange A[i] ↔ A[largest]
          HEAPIFY(A, largest)
```

Figure 7.2 illustrates the action of HEAPIFY. At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ is determined, and its index is stored in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is a heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes node i and its children to satisfy the heap property. The node *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at *largest* may violate the heap property. Consequently, HEAPIFY must be called recursively on that subtree.

The running time of HEAPIFY on a subtree of size n rooted at given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run HEAPIFY on a subtree rooted at one of the children of node i. The children's subtrees each have size at most $2n/3$—the worst case occurs when the last row of the tree is exactly half full—and the running time of HEAPIFY can therefore be described by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1) .$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of HEAPIFY on a node of height h as $O(h)$.

Exercises

7.2-1
Using Figure 7.2 as a model, illustrate the operation of HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

7.2-2
What is the effect of calling HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

7.2-3
What is the effect of calling HEAPIFY(A, i) for $i > \textit{heap-size}[A]/2$?

7.2-4
The code for HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

7.2-5
Show that the worst-case running time of HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause HEAPIFY to be called recursively at every node on a path from the root down to a leaf.)

7.3 Building a heap

We can use the procedure HEAPIFY in a bottom-up manner to convert an array $A[1\,..\,n]$, where $n = length[A]$, into a heap. Since the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)\,..\,n]$ are all leaves of the tree, each is a 1-element heap to begin with. The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each one. The order in which the nodes are processed guarantees that the subtrees rooted at children of a node i are heaps before HEAPIFY is run at that node.

```
BUILD-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
     do HEAPIFY(A, i)
```

Figure 7.3 shows an example of the action of BUILD-HEAP.

We can compute a simple upper bound on the running time of BUILD-HEAP as follows. Each call to HEAPIFY costs $O(\lg n)$ time, and there are $O(n)$ such calls. Thus, the running time is at most $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the property that in an n-element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 7.3-3).

The time required by HEAPIFY when called on a node of height h is $O(h)$, so we can express the total cost of BUILD-HEAP as

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) . \tag{7.2}$$

The last summation can be evaluated by substituting $x = 1/2$ in the formula (3.6), which yields

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, the running time of BUILD-HEAP can be bounded as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

Hence, we can build a heap from an unordered array in linear time.

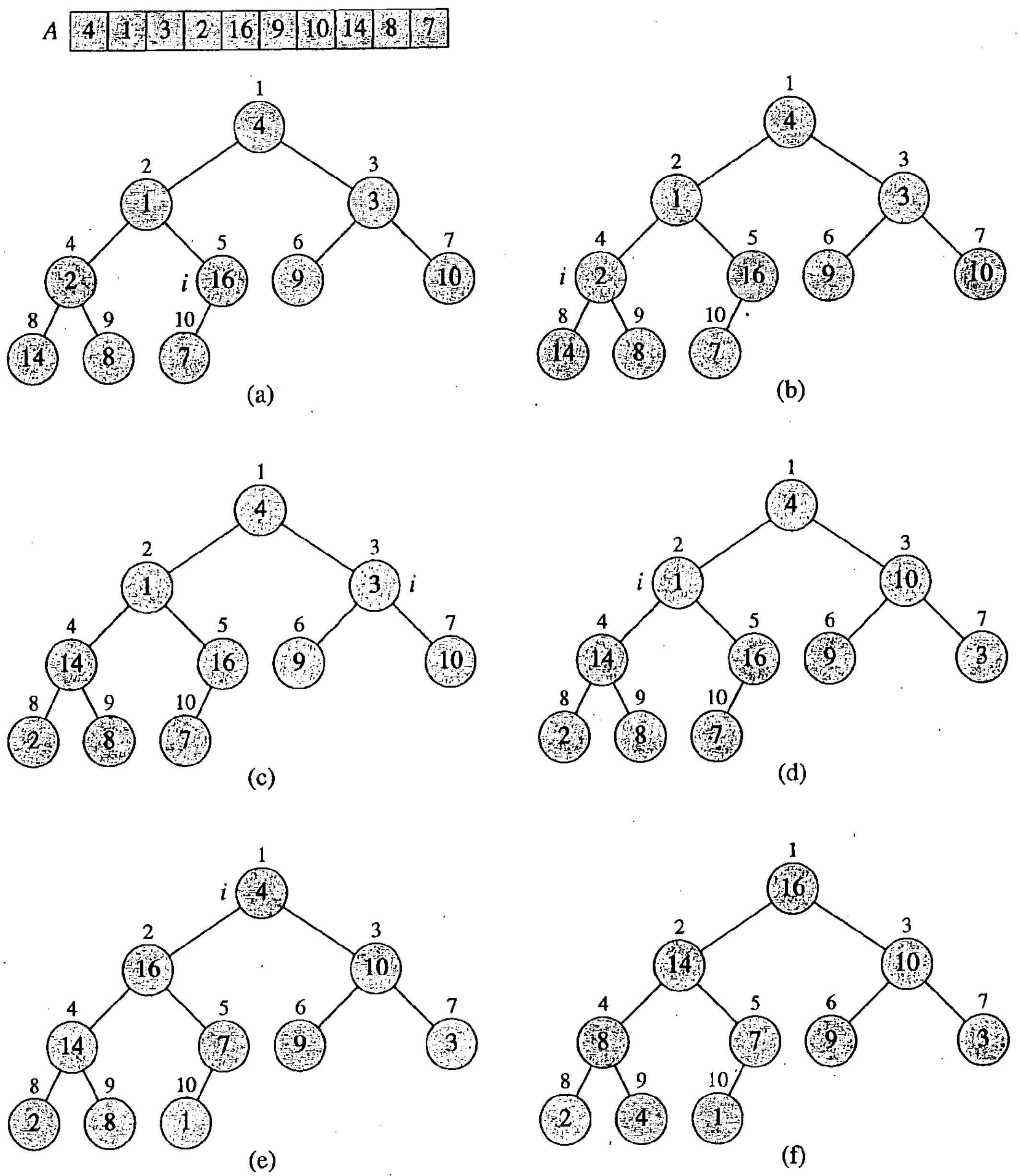


Figure 7.3 The operation of BUILD-HEAP, showing the data structure before the call to HEAPIFY in line 3 of BUILD-HEAP. (**a**) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i points to node 5 before the call HEAPIFY(A, i). (**b**) The data structure that results. The loop index i for the next iteration points to node 4. (**c**)–(**e**) Subsequent iterations of the **for** loop in BUILD-HEAP. Observe that whenever HEAPIFY is called on a node, the two subtrees of that node are both heaps. (**f**) The heap after BUILD-HEAP finishes.

Exercises

7.3-1

Using Figure 7.3 as a model, illustrate the operation of Build-Heap on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

7.3-2

Why do we want the loop index i in line 2 of Build-Heap to decrease from $\lfloor length[A]/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor length[A]/2 \rfloor$?

7.3-3

Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

7.4 The heapsort algorithm

The heapsort algorithm starts by using Build-Heap to build a heap on the input array $A[1\,..\,n]$, where $n = length[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If we now "discard" node n from the heap (by decrementing *heap-size*$[A]$), we observe that $A[1\,..\,(n-1)]$ can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap property (7.1). All that is needed to restore the heap property, however, is one call to Heapify$(A, 1)$, which leaves a heap in $A[1\,..\,(n-1)]$. The heapsort algorithm then repeats this process for the heap of size $n - 1$ down to a heap of size 2.

```
Heapsort(A)
1  Build-Heap(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          Heapify(A, 1)
```

Figure 7.4 shows an example of the operation of heapsort after the heap is initially built. Each heap is shown at the beginning of an iteration of the **for** loop in line 2.

The Heapsort procedure takes time $O(n \lg n)$, since the call to Build-Heap takes time $O(n)$ and each of the $n - 1$ calls to Heapify takes time $O(\lg n)$.

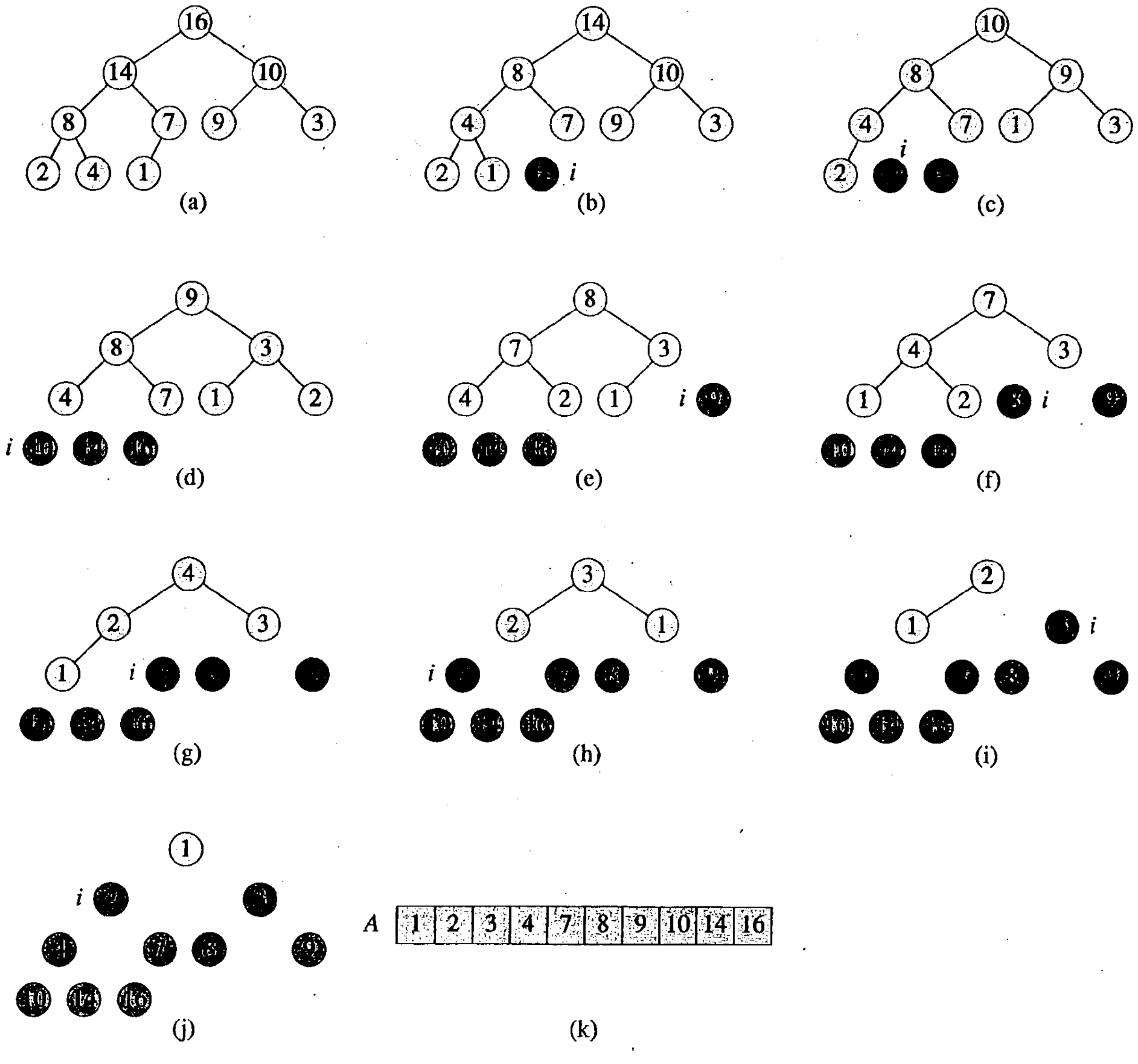


Figure 7.4 The operation of HEAPSORT. **(a)** The heap data structure just after it has been built by BUILD-HEAP. **(b)–(j)** The heap just after each call of HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Exercises

7.4-1

Using Figure 7.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

7.4-2

What is the running time of heapsort on an array A of length n that is already sorted in increasing order? What about decreasing order?

7.4-3

Show that the running time of heapsort is $\Omega(n \lg n)$.

7.5 Priority queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, presented in Chapter 8, usually beats it in practice. Nevertheless, the heap data structure itself has enormous utility. In this section, we present one of the most popular applications of a heap: its use as an efficient priority queue.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A priority queue supports the following operations.

INSERT(S, x) inserts the element x into the set S. This operation could be written as $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

One application of priority queues is to schedule jobs on a shared computer. The priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

A priority queue can also be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. For this application, it is natural to reverse the linear order of the priority queue and support the operations MINIMUM and EXTRACT-MIN instead of MAXIMUM and EXTRACT-MAX. The simulation program uses EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, they are inserted into the priority queue using INSERT.

Not surprisingly, we can use a heap to implement a priority queue. The operation HEAP-MAXIMUM returns the maximum heap element in $\Theta(1)$ time by simply returning the value $A[1]$ in the heap. The HEAP-EXTRACT-MAX procedure is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure:

```
HEAP-EXTRACT-MAX($A$)
if $heap\text{-}size[A] < 1$
    then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[heap\text{-}size[A]]$
$heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
HEAPIFY($A, 1$)
return $max$
```

The running time of HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for HEAPIFY.

The HEAP-INSERT procedure inserts a node into heap A. To do so, it first expands the heap by adding a new leaf to the tree. Then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT from Section 1.1, it traverses a path from this leaf toward the root to find a proper place for the new element.

```
HEAP-INSERT($A, key$)
$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$i \leftarrow heap\text{-}size[A]$
while $i > 1$ and $A[\text{PARENT}(i)] < key$
    do $A[i] \leftarrow A[\text{PARENT}(i)]$
       $i \leftarrow \text{PARENT}(i)$
$A[i] \leftarrow key$
```

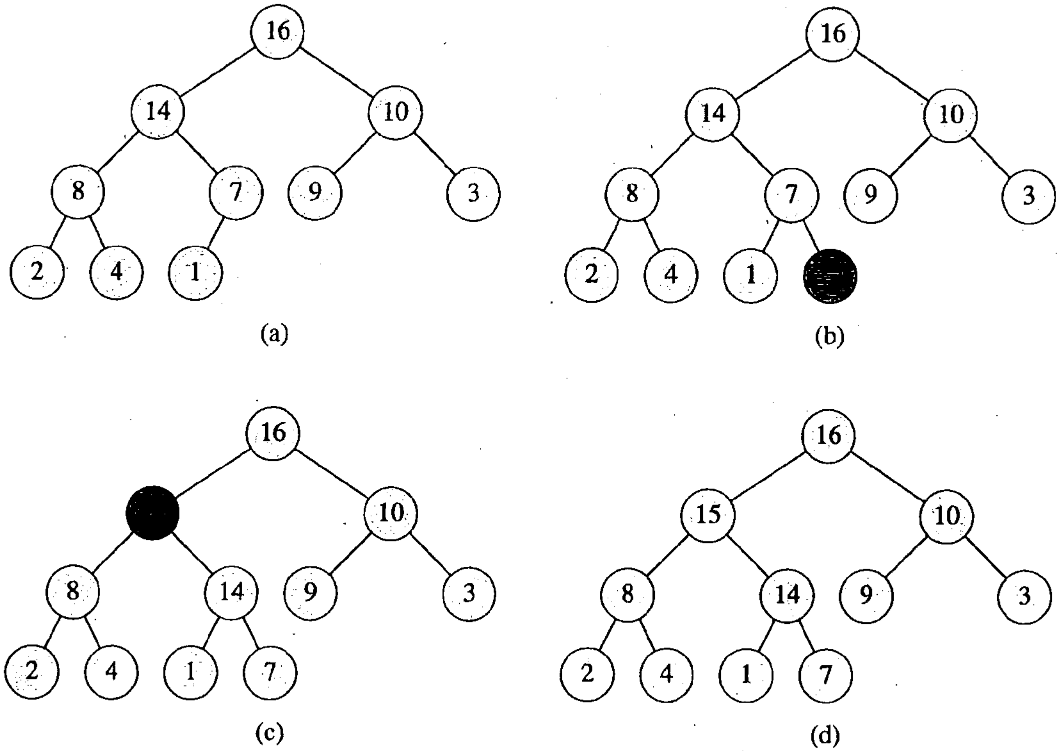
Figure 7.5 shows an example of a HEAP-INSERT operation. The running time of HEAP-INSERT on an n-element heap is $O(\lg n)$, since the path traced from the new leaf to the root has length $O(\lg n)$.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.

Exercises

7.5-1
Using Figure 7.5 as a model, illustrate the operation of HEAP-INSERT($A, 3$) on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

7.5-2
Illustrate the operation of HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.



(a) (b) (c) (d)

Figure 7.5 The operation of Heap-Insert. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

7.5-3

Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (FIFO's and stacks are defined in Section 11.1.)

7.5-4

Give an $O(\lg n)$-time implementation of the procedure Heap-Increase-Key(A, i, k), which sets $A[i] \leftarrow \max(A[i], k)$ and updates the heap structure appropriately.

7.5-5

The operation Heap-Delete(A, i) deletes the item in node i from heap A. Give an implementation of Heap-Delete that runs in $O(\lg n)$ time for an n-element heap.

7.5-6

Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a heap for k-way merging.)

Problems

7-1 Building a heap using insertion

The procedure Build-Heap in Section 7.3 can be implemented by repeatedly using Heap-Insert to insert the elements into the heap. Consider the following implementation:

```
Build-Heap'(A)
heap-size[A] ← 1
for i ← 2 to length[A]
    do Heap-Insert(A, A[i])
```

a. Do the procedures Build-Heap and Build-Heap$'$ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, Build-Heap$'$ requires $\Theta(n \lg n)$ time to build an n-element heap.

7-2 Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but instead of 2 children, nodes have d children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of Extract-Max. Analyze its running time in terms of d and n.

d. Give an efficient implementation of Insert. Analyze its running time in terms of d and n.

e. Give an efficient implementation of Heap-Increase-Key(A, i, k), which sets $A[i] \leftarrow \max(A[i], k)$ and updates the heap structure appropriately. Analyze its running time in terms of d and n.

Chapter notes

The heapsort algorithm was invented by Williams [202], who also described how to implement a priority queue with a heap. The Build-Heap procedure was suggested by Floyd [69].

8 Quicksort

Quicksort is a sorting algorithm whose worst-case running time is $\Theta(n^2)$ on an input array of n numbers. In spite of this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\Theta(n \lg n)$, and the constant factors hidden in the $\Theta(n \lg n)$ notation are quite small. It also has the advantage of sorting in place (see page 3), and it works well even in virtual memory environments.

Section 8.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we start with an intuitive discussion of its performance in Section 8.2 and postpone its precise analysis to the end of the chapter. Section 8.3 presents two versions of quicksort that use a random-number generator. These "randomized" algorithms have many desirable properties. Their average-case running time is good, and no particular input elicits their worst-case behavior. One of the randomized versions of quicksort is analyzed in Section 8.4, where it is shown to run in $O(n^2)$ time in the worst case and in $O(n \lg n)$ time on average.

8.1 Description of quicksort

Quicksort, like merge sort, is based on the divide-and-conquer paradigm introduced in Section 1.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$.

Divide: The array $A[p..r]$ is partitioned (rearranged) into two nonempty subarrays $A[p..q]$ and $A[q+1..r]$ such that each element of $A[p..q]$ is less than or equal to each element of $A[q+1..r]$. The index q is computed as part of this partitioning procedure.

Conquer: The two subarrays $A[p..q]$ and $A[q+1..r]$ are sorted by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

The following procedure implements quicksort.

```
QUICKSORT($A, p, r$)
if $p < r$
  then $q \leftarrow$ PARTITION($A, p, r$)
       QUICKSORT($A, p, q$)
       QUICKSORT($A, q + 1, r$)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, *length*[A]).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION($A, p, r$)
 1  $x \leftarrow A[p]$
 2  $i \leftarrow p - 1$
 3  $j \leftarrow r + 1$
 4  while TRUE
 5      do repeat $j \leftarrow j - 1$
 6           until $A[j] \leq x$
 7         repeat $i \leftarrow i + 1$
 8           until $A[i] \geq x$
 9         if $i < j$
10           then exchange $A[i] \leftrightarrow A[j]$
11           else return $j$
```

Figure 8.1 shows how PARTITION works. It first selects an element $x = A[p]$ from $A[p \,..\, r]$ as a "pivot" element around which to partition $A[p \,..\, r]$. It then grows two regions $A[p \,..\, i]$ and $A[j \,..\, r]$ from the top and bottom of $A[p \,..\, r]$, respectively, such that every element in $A[p \,..\, i]$ is less than or equal to x and every element in $A[j \,..\, r]$ is greater than or equal to x. Initially, $i = p - 1$ and $j = r + 1$, so the two regions are empty.

Within the body of the **while** loop, the index j is decremented and the index i is incremented, in lines 5–8, until $A[i] \geq x \geq A[j]$. Assuming that these inequalities are strict, $A[i]$ is too large to belong to the bottom region and $A[j]$ is too small to belong to the top region. Thus, by exchanging $A[i]$ and $A[j]$ as is done in line 10, we can extend the two regions. (If the inequalities are not strict, the exchange can be performed anyway.)

The body of the **while** loop repeats until $i \geq j$, at which point the entire array $A[p \,..\, r]$ has been partitioned into two subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$, where $p \leq q < r$, such that no element of $A[p \,..\, q]$ is larger than any element of $A[q + 1 \,..\, r]$. The value $q = j$ is returned at the end of the procedure.

Conceptually, the partitioning procedure performs a simple function: it puts elements smaller than x into the bottom region of the array and

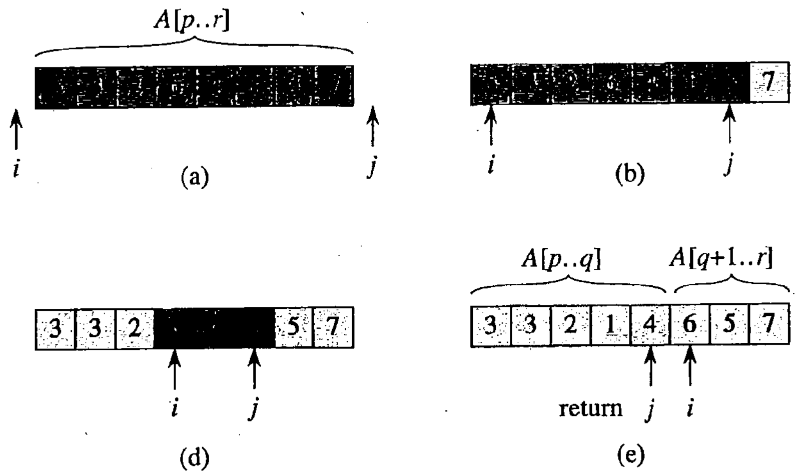


Figure 8.1 The operation of PARTITION on a sample array. Lightly shaded array elements have been placed into the correct partitions, and heavily shaded elements are not yet in their partitions. **(a)** The input array, with the initial values of i and j just off the left and right ends of the array. We partition around $x = A[p] = 5$. **(b)** The positions of i and j at line 9 of the first iteration of the **while** loop. **(c)** The result of exchanging the elements pointed to by i and j in line 10. **(d)** The positions of i and j at line 9 of the second iteration of the **while** loop. **(e)** The positions of i and j at line 9 of the third and last iteration of the **while** loop. The procedure terminates because $i \geq j$, and the value $q = j$ is returned. Array elements up to and including $A[j]$ are less than or equal to $x = 5$, and array elements after $A[j]$ are greater than or equal to $x = 5$.

elements larger than x into the top region. There are technicalities that make the pseudocode of PARTITION a little tricky, however. For example, the indices i and j never index the subarray $A[p\,..\,r]$ out of bounds, but this isn't entirely apparent from the code. As another example, it is important that $A[p]$ be used as the pivot element x. If $A[r]$ is used instead and it happens that $A[r]$ is also the largest element in the subarray $A[p\,..\,r]$, then PARTITION returns to QUICKSORT the value $q = r$, and QUICKSORT loops forever. Problem 8-1 asks you to prove PARTITION correct.

The running time of PARTITION on an array $A[p\,..\,r]$ is $\Theta(n)$, where $n = r - p + 1$ (see Exercise 8.1-3).

Exercises

8.1-1

Using Figure 8.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

8.1-2

What value of q does PARTITION return when all elements in the array $A[p\,..\,r]$ have the same value?

8.1-3
Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

8.1-4
How would you modify QUICKSORT to sort in nonincreasing order?

8.2 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slow as insertion sort. In this section, we shall informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one region with $n-1$ elements and one with only 1 element. (This claim is proved in Section 8.4.1.) Let us assume that this unbalanced partitioning arises at every step of the algorithm. Since partitioning costs $\Theta(n)$ time and $T(1) = \Theta(1)$, the recurrence for the running time is

$$T(n) = T(n-1) + \Theta(n) .$$

To evaluate this recurrence, we observe that $T(1) = \Theta(1)$ and then iterate:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) . \end{aligned}$$

We obtain the last line by observing that $\sum_{k=1}^{n} k$ is the arithmetic series (3.2). Figure 8.2 shows a recursion tree for this worst-case execution of quicksort. (See Section 4.2 for a discussion of recursion trees.)

Thus, if the partitioning is maximally unbalanced at every recursive step of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already

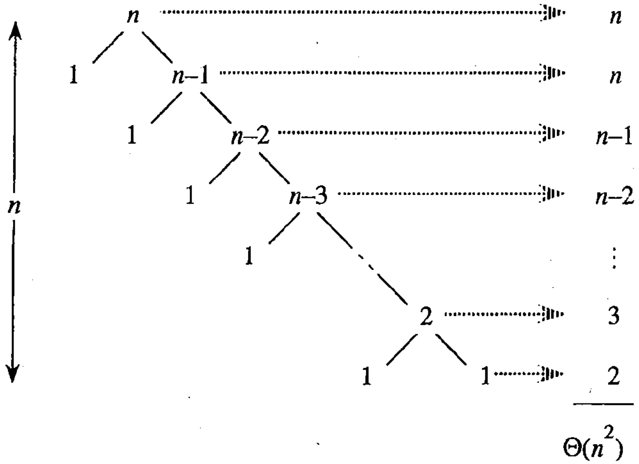


Figure 8.2 A recursion tree for QUICKSORT in which the PARTITION procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$.

completely sorted—a common situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

If the partitioning procedure produces two regions of size $n/2$, quicksort runs much faster. The recurrence is then

$$T(n) = 2T(n/2) + \Theta(n) ,$$

which by case 2 of the master theorem (Theorem 4.1) has solution $T(n) = \Theta(n \lg n)$. Thus, this best-case partitioning produces a much faster algorithm. Figure 8.3 shows the recursion tree for this best-case execution of quicksort.

Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to the worst case, as the analyses in Section 8.4 will show. The key to understanding why this might be true is to understand how the balance of the partitioning is reflected in the recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + n$$

on the running time of quicksort, where we have replaced $\Theta(n)$ by n for convenience. Figure 8.4 shows the recursion tree for this recurrence. No-

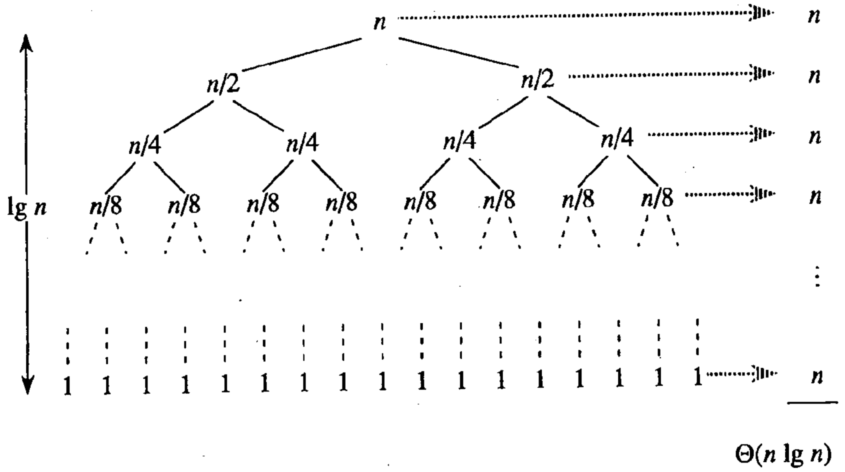


Figure 8.3 A recursion tree for Quicksort in which Partition always balances the two sides of the partition equally (the best case). The resulting running time is $\Theta(n \lg n)$.

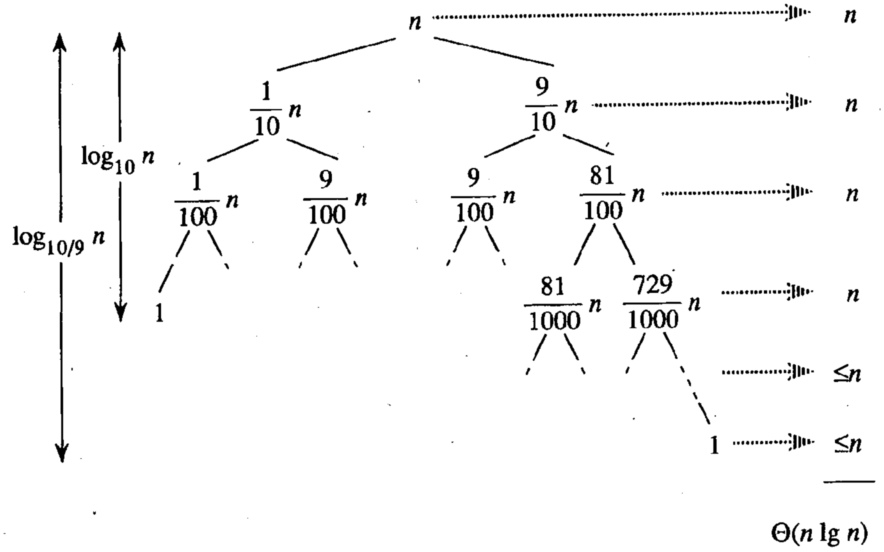


Figure 8.4 A recursion tree for Quicksort in which Partition always produces a 9-to-1 split, yielding a running time of $\Theta(n \lg n)$.

tice that every level of the tree has cost n, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most n. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. The total cost of quicksort is therefore $\Theta(n \lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $\Theta(n \lg n)$ time—asymptotically the same as if the split were right down the middle. In fact, even a 99-to-1 split yields an $O(n \lg n)$ running time. The reason is that any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $\Theta(n \lg n)$ whenever the split has constant proportionality.

Intuition for the average case

To develop a clear notion of the average case for quicksort, we must make an assumption about how frequently we expect to encounter the various inputs. A common assumption is that all permutations of the input numbers are equally likely. We shall discuss this assumption in the next section, but first let's explore its ramifications.

When we run quicksort on a random input array, it is unlikely that the partitioning always happens in the same way at every level, as our informal analysis has assumed. We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 8.2-5 asks to you show that about 80 percent of the time PARTITION produces a split that is more balanced than 9 to 1, and about 20 percent of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose for the sake of intuition, however, that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 8.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is n for partitioning and the subarrays produced have sizes $n - 1$ and 1: the worst case. At the next level, the subarray of size $n-1$ is best-case partitioned into two subarrays of size $(n - 1)/2$. Let's assume that the boundary-condition cost is 1 for the subarray of size 1.

The combination of the bad split followed by the good split produces three subarrays of sizes 1, $(n - 1)/2$, and $(n - 1)/2$ at a combined cost of $2n - 1 = \Theta(n)$. Certainly, this situation is no worse than that in Figure 8.5(b), namely a single level of partitioning that produces two subarrays of sizes $(n - 1)/2 + 1$ and $(n - 1)/2$ at a cost of $n = \Theta(n)$. Yet this latter situation is very nearly balanced, certainly better than 9 to 1. Intuitively, the $\Theta(n)$ cost of the bad split can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quick-

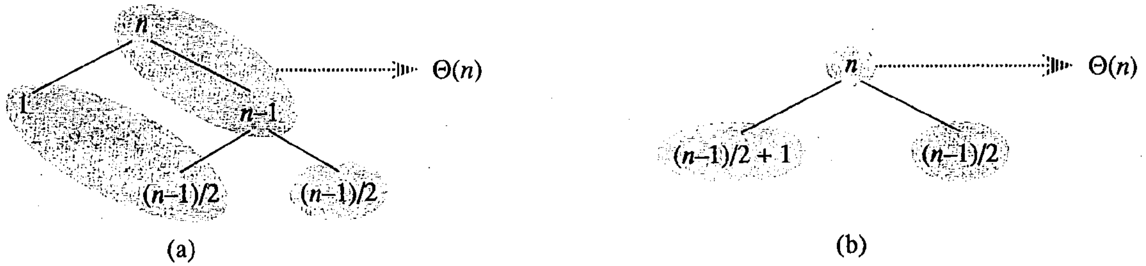


Figure 8.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 1 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: two subarrays of size $(n - 1)/2$. **(b)** A single level of a recursion tree that is worse than the combined levels in (a), yet very well balanced.

sort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the O-notation. We shall give a rigorous analysis of the average case in Section 8.4.2.

Exercises

8.2-1

Show that the running time of QUICKSORT is $\Theta(n \lg n)$ when all elements of array A have the same value.

8.2-2

Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A is sorted in nonincreasing order.

8.2-3

Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

8.2-4

Suppose that the splits at every level of quicksort are in the proportion $1 - \alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1 - \alpha)$. (Don't worry about integer round-off.)

8.2-5 ★

Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that on a random input array, PARTITION produces a split more balanced than $1 - \alpha$ to α. For what value of α are the odds even that the split is more balanced than less balanced?

8.3 Randomized versions of quicksort

In exploring the average-case behavior of quicksort, we have made an assumption that all permutations of the input numbers are equally likely. When this assumption on the distribution of the inputs is valid, many people regard quicksort as the algorithm of choice for large enough inputs. In an engineering situation, however, we cannot always expect it to hold. (See Exercise 8.2-3.) This section introduces the notion of a randomized algorithm and presents two randomized versions of quicksort that overcome the assumption that all permutations of the input numbers are equally likely.

An alternative to *assuming* a distribution of inputs is to *impose* a distribution. For example, suppose that before sorting the input array, quicksort randomly permutes the elements to enforce the property that every permutation is equally likely. (Exercise 8.3-4 asks for an algorithm that randomly permutes the elements of an array of size n in time $O(n)$.) This modification does not improve the worst-case running time of the algorithm, but it does make the running time independent of the input ordering.

We call an algorithm ***randomized*** if its behavior is determined not only by the input but also by values produced by a ***random-number generator***. We shall assume that we have at our disposal a random-number generator RANDOM. A call to RANDOM(a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM$(0, 1)$ produces a 0 with probability 1/2 and a 1 with probability 1/2. Each integer returned by RANDOM is independent of the integers returned on previous calls. You may imagine RANDOM as rolling a $(b - a + 1)$-sided die to obtain its output. (In practice, most programming environments offer a ***pseudorandom-number generator***: a deterministic algorithm that returns numbers that "look" statistically random.)

This randomized version of quicksort has an interesting property that is also possessed by many other randomized algorithms: *no particular input elicits its worst-case behavior.* Instead, its worst case depends on the random-number generator. Even intentionally, you cannot produce a bad input array for quicksort, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an unlucky permutation to be sorted. Exercise 13.4-4 shows that almost all permutations cause quicksort to perform nearly as well as the average case: there are *very* few permutations that cause near-worst-case behavior.

A randomized strategy is typically useful when there are many ways in which an algorithm can proceed but it is difficult to determine a way that is guaranteed to be good. If many of the alternatives are good, simply choosing one randomly can yield a good strategy. Often, an algorithm must make many choices during its execution. If the benefits of good choices outweigh the costs of bad choices, a random selection of good and bad choices can yield an efficient algorithm. We noted in Section 8.2 that a mixture of good and bad splits yields a good running time for quicksort, and thus it makes sense that randomized versions of the algorithm should perform well.

By modifying the PARTITION procedure, we can design another randomized version of quicksort that uses this random-choice strategy. At each step of the quicksort algorithm, before the array is partitioned, we exchange element $A[p]$ with an element chosen at random from $A[p\,..\,r]$. This modification ensures that the pivot element $x = A[p]$ is equally likely to be any of the $r - p + 1$ elements in the subarray. Thus, we expect the split of the input array to be reasonably well balanced on average. The randomized algorithm based on randomly permuting the input array also works well on average, but it is somewhat more difficult to analyze than this version.

The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning:

```
RANDOMIZED-PARTITION($A, p, r$)
  $i \leftarrow$ RANDOM($p, r$)
  exchange $A[p] \leftrightarrow A[i]$
  return PARTITION($A, p, r$)
```

We now make the new quicksort call RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT($A, p, r$)
  if $p < r$
    then $q \leftarrow$ RANDOMIZED-PARTITION($A, p, r$)
         RANDOMIZED-QUICKSORT($A, p, q$)
         RANDOMIZED-QUICKSORT($A, q + 1, r$)
```

We analyze this algorithm in the next section.

Exercises

8.3-1

Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance?

8.3-2

During the running of the procedure RANDOMIZED-QUICKSORT, how many calls are made to the random-number generator RANDOM in the worst case? How does the answer change in the best case?

8.3-3 ⋆

Describe an implementation of the procedure RANDOM(a, b) that uses only fair coin flips. What is the expected running time of your procedure?

8.3-4 ⋆

Give a $\Theta(n)$-time, randomized procedure that takes as input an array $A[1\,..\,n]$ and performs a random permutation on the array elements.

8.4 Analysis of quicksort

Section 8.2 gave some intuition for the worst-case behavior of quicksort and for why we expect it to run quickly. In this section, we analyze the behavior of quicksort more rigorously. We begin with a worst-case analysis, which applies to either QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an average-case analysis of RANDOMIZED-QUICKSORT.

8.4.1 Worst-case analysis

We saw in Section 8.2 that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

Using the substitution method (see Section 4.1), we can show that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) , \tag{8.1}$$

where the parameter q ranges from 1 to $n-1$ because the procedure PARTITION produces two regions, each having size at least 1. We guess that $T(n) \le cn^2$ for some constant c. Substituting this guess into (8.1), we obtain

$$\begin{aligned} T(n) &\le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) \\ &= c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n) . \end{aligned}$$

The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints, as can be seen since the second derivative of the expression with respect to q is positive (see Exercise 8.4-2). This gives us the bound $\max_{1 \le q \le n-1}(q^2 + (n-q)^2) \le 1^2 + (n-1)^2 = n^2 - 2(n-1)$.

Continuing with our bounding of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le cn^2 - 2c(n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

since we can pick the constant c large enough so that the $2c(n-1)$ term dominates the $\Theta(n)$ term. Thus, the (worst-case) running time of quicksort is $\Theta(n^2)$.

8.4.2 Average-case analysis

We have already given an intuitive argument why the average-case running time of RANDOMIZED-QUICKSORT is $\Theta(n \lg n)$: if the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$ and $\Theta(n)$ work is performed at $\Theta(\lg n)$ of these levels. We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates. We can then develop a recurrence for the average time required to sort an n-element array and solve this recurrence to determine bounds on the expected running time. As part of the process of solving the recurrence, we shall develop tight bounds on an interesting summation.

Analysis of partitioning

We first make some observations about the operation of PARTITION. When PARTITION is called in line 3 of the procedure RANDOMIZED-PARTITION, the element $A[p]$ has already been exchanged with a random element in $A[p\,..\,r]$. To simplify the analysis, we assume that all input numbers are distinct. If all input numbers are not distinct, it is still true that quicksort's average-case running time is $O(n \lg n)$, but a somewhat more intricate analysis than we present here is required.

Our first observation is that the value of q returned by PARTITION depends only on the rank of $x = A[p]$ among the elements in $A[p\,..\,r]$. (The ***rank*** of a number in a set is the number of elements less than or equal to it.) If we let $n = r - p + 1$ be the number of elements in $A[p\,..\,r]$, swapping $A[p]$ with a random element from $A[p\,..\,r]$ yields a probability $1/n$ that $\text{rank}(x) = i$ for $i = 1, 2, \ldots, n$.

We next compute the likelihoods of the various outcomes of the partitioning. If $\text{rank}(x) = 1$, then the first time through the **while** loop in lines 4–11 of PARTITION, index i stops at $i = p$ and index j stops at $j = p$. Thus, when $q = j$ is returned, the "low" side of the partition contains the sole element $A[p]$. This event occurs with probability $1/n$ since that is the probability that $\text{rank}(x) = 1$.

If $\text{rank}(x) \ge 2$, then there is at least one element smaller than $x = A[p]$. Consequently, the first time through the **while** loop, index i stops at $i = p$

but j stops before reaching p. An exchange with $A[p]$ is then made to put $A[p]$ in the high side of the partition. When PARTITION terminates, each of the $\text{rank}(x) - 1$ elements in the low side of the partition is strictly less than x. Thus, for each $i = 1, 2, \ldots, n-1$, when $\text{rank}(x) \geq 2$, the probability is $1/n$ that the low side of the partition has i elements.

Combining these two cases, we conclude that the size $q - p + 1$ of the low side of the partition is 1 with probability $2/n$ and that the size is i with probability $1/n$ for $i = 2, 3, \ldots, n-1$.

A recurrence for the average case

We now establish a recurrence for the expected running time of RANDOMIZED-QUICKSORT. Let $T(n)$ denote the average time required to sort an n-element input array. A call to RANDOMIZED-QUICKSORT with a 1-element array takes constant time, so we have $T(1) = \Theta(1)$. A call to RANDOMIZED-QUICKSORT with an array $A[1\,..\,n]$ of length n uses time $\Theta(n)$ to partition the array. The PARTITION procedure returns an index q, and then RANDOMIZED-QUICKSORT is called recursively with subarrays of length q and $n - q$. Consequently, the average time to sort an array of length n can be expressed as

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) . \qquad (8.2)$$

The value of q has an almost uniform distribution, except that the value $q = 1$ is twice as likely as the others, as was noted above. Using the facts that $T(1) = \Theta(1)$ and $T(n-1) = O(n^2)$ from our worst-case analysis, we have

$$\begin{aligned}\frac{1}{n}(T(1) + T(n-1)) &= \frac{1}{n}(\Theta(1) + O(n^2)) \\ &= O(n) ,\end{aligned}$$

and the term $\Theta(n)$ in equation (8.2) can therefore absorb the expression $\frac{1}{n}(T(1) + T(n-1))$. We can thus restate recurrence (8.2) as

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) . \qquad (8.3)$$

Observe that for $k = 1, 2, \ldots, n-1$, each term $T(k)$ of the sum occurs once as $T(q)$ and once as $T(n-q)$. Collapsing the two terms of the sum yields

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) . \qquad (8.4)$$

Solving the recurrence

We can solve the recurrence (8.4) using the substitution method. Assume inductively that $T(n) \le an \lg n + b$ for some constants $a > 0$ and $b > 0$ to be determined. We can pick a and b sufficiently large so that $an \lg n + b$ is greater than $T(1)$. Then for $n > 1$, we have by substitution

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\le \frac{2}{n}\sum_{k=1}^{n-1}(ak \lg k + b) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n) . \end{aligned}$$

We show below that the summation in the last line can be bounded by

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \tag{8.5}$$

Using this bound, we obtain

$$\begin{aligned} T(n) &\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\ &\le an \lg n - \frac{a}{4}n + 2b + \Theta(n) \\ &= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \\ &\le an \lg n + b , \end{aligned}$$

since we can choose a large enough so that $\frac{a}{4}n$ dominates $\Theta(n) + b$. We conclude that quicksort's average running time is $O(n \lg n)$.

Tight bounds on the key summation

It remains to prove the bound (8.5) on the summation

$$\sum_{k=1}^{n-1} k \lg k .$$

Since each term is at most $n \lg n$, we have the bound

$$\sum_{k=1}^{n-1} k \lg k \le n^2 \lg n ,$$

which is tight to within a constant factor. This bound is not strong enough to solve the recurrence as $T(n) = O(n \lg n)$, however. Specifically, we need a bound of $\frac{1}{2}n^2 \lg n - \Omega(n^2)$ for the solution of the recurrence to work out.

We can get this bound on the summation by splitting it into two parts, as discussed in Section 3.2 on page 48. We obtain

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k .$$

The $\lg k$ in the first summation on the right is bounded above by $\lg(n/2) = \lg n - 1$. The $\lg k$ in the second summation is bounded above by $\lg n$. Thus,

$$\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &\le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\le \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&\le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}$$

if $n \ge 2$. This is the bound (8.5).

Exercises

8.4-1

Show that quicksort's best-case running time is $\Omega(n \lg n)$.

8.4-2

Show that $q^2 + (n-q)^2$ achieves a maximum over $q = 1, 2, \ldots, n-1$ when $q = 1$ or $q = n - 1$.

8.4-3

Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

8.4-4

The running time of quicksort can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should k be picked, both in theory and in practice?

8.4-5 ⋆

Prove the identity

$$\int x \ln x \, dx = \frac{1}{2} x^2 \ln x - \frac{1}{4} x^2 ,$$

and then use the integral approximation method to give a tighter upper bound than (8.5) on the summation $\sum_{k=1}^{n-1} k \lg k$.

8.4-6 ⋆

Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median. Approximate the probability of getting at worst an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

Problems

8-1 Partition correctness

Give a careful argument that the procedure PARTITION in Section 8.1 is correct. Prove the following:

a. The indices i and j never reference an element of A outside the interval $[p \mathinner{..} r]$.

b. The index j is not equal to r when PARTITION terminates (so that the split is always nontrivial).

c. Every element of $A[p \mathinner{..} j]$ is less than or equal to every element of $A[j+1 \mathinner{..} r]$ when PARTITION terminates.

8-2 Lomuto's partitioning algorithm

Consider the following variation of PARTITION, due to N. Lomuto. To partition $A[p \mathinner{..} r]$, this version grows two regions, $A[p \mathinner{..} i]$ and $A[i+1 \mathinner{..} j]$, such that every element in the first region is less than or equal to $x = A[r]$ and every element in the second region is greater than x.

```
LOMUTO-PARTITION($A, p, r$)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r$
    do if $A[j] \leq x$
          then $i \leftarrow i + 1$
               exchange $A[i] \leftrightarrow A[j]$
if $i < r$
   then return $i$
   else return $i - 1$
```

a. Argue that LOMUTO-PARTITION is correct.

b. What are the maximum numbers of times that an element can be moved by PARTITION and by LOMUTO-PARTITION?

c. Argue that LOMUTO-PARTITION, like PARTITION, runs in $\Theta(n)$ time on an n-element subarray.

d. How does replacing Partition by Lomuto-Partition affect the running time of Quicksort when all input values are equal?

e. Define a procedure Randomized-Lomuto-Partition that exchanges $A[r]$ with a randomly chosen element in $A[p \mathinner{..} r]$ and then calls Lomuto-Partition. Show that the probability that a given value q is returned by Randomized-Lomuto-Partition is equal to the probability that $p + r - q$ is returned by Randomized-Partition.

8-3 Stooge sort

Professors Howard, Fine, and Howard have proposed the following "elegant" sorting algorithm:

```
Stooge-Sort($A, i, j$)
if $A[i] > A[j]$
    then exchange $A[i] \leftrightarrow A[j]$
if $i + 1 \geq j$
    then return
$k \leftarrow \lfloor (j - i + 1)/3 \rfloor$        ▷ Round down.
Stooge-Sort($A, i, j - k$)      ▷ First two-thirds.
Stooge-Sort($A, i + k, j$)      ▷ Last two-thirds.
Stooge-Sort($A, i, j - k$)      ▷ First two-thirds again.
```

a. Argue that Stooge-Sort$(A, 1, \mathit{length}[A])$ correctly sorts the input array $A[1 \mathinner{..} n]$, where $n = \mathit{length}[A]$.

b. Give a recurrence for the worst-case running time of Stooge-Sort and a tight asymptotic (Θ-notation) bound on the worst-case running time.

c. Compare the worst-case running time of Stooge-Sort with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

8-4 Stack depth for quicksort

The Quicksort algorithm of Section 8.1 contains two recursive calls to itself. After the call to Partition, the left subarray is recursively sorted and then the right subarray is recursively sorted. The second recursive call in Quicksort is not really necessary; it can be avoided by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion.

```
QUICKSORT′$(A, p, r)$
while $p < r$
    do ▷ Partition and sort left subarray
        $q \leftarrow$ PARTITION$(A, p, r)$
        QUICKSORT′$(A, p, q)$
        $p \leftarrow q + 1$
```

a. Argue that QUICKSORT′$(A, 1, \mathit{length}[A])$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are actually represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which the stack depth of QUICKSORT′ is $\Theta(n)$ on an n-element input array.

c. Modify the code for QUICKSORT′ so that the worst-case stack depth is $\Theta(\lg n)$.

8-5 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around an element x that is chosen more carefully than by picking a random element from the subarray. One common approach is the ***median-of-3*** method: choose x as the median (middle element) of a set of 3 elements randomly selected from the subarray. For this problem, let us assume that the elements in the input array $A[1 \,.\,.\, n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1 \,.\,.\, n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = A'[i]\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n - 1$. (Note that $p_1 = p_n = 0$.)

b. By what amount have we increased the likelihood of choosing $x = A'[\lfloor (n + 1)/2 \rfloor]$, the median of $A[1 \,.\,.\, n]$, compared to the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. If we define a "good" split to mean choosing $x = A'[i]$, where $n/3 \leq i \leq 2n/3$, by what amount have we increased the likelihood of getting a good split compared to the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

d. Argue that the median-of-3 method affects only the constant factor in the $\Omega(n \lg n)$ running time of quicksort.

Chapter notes

The quicksort procedure was invented by Hoare [98]. Sedgewick [174] provides a good reference on the details of implementation and how they matter. The advantages of randomized algorithms were articulated by Rabin [165].

9 Sorting in Linear Time

We have now introduced several algorithms that can sort n numbers in $O(n \lg n)$ time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements.* We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 9.1, we shall prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort a sequence of n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 9.2, 9.3, and 9.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time. Needless to say, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

9.1 Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all of the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about

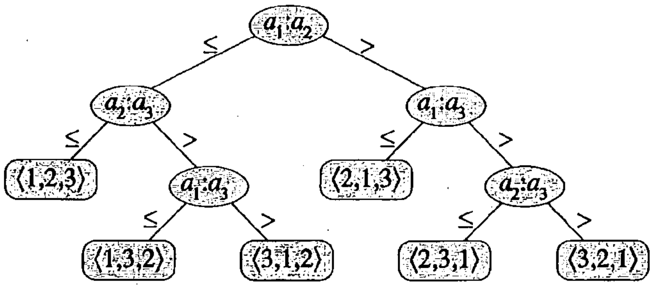


Figure 9.1 The decision tree for insertion sort operating on three elements. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \le a_j$.

The decision-tree model

Comparison sorts can be viewed abstractly in terms of ***decision trees.*** A decision tree represents the comparisons performed by a sorting algorithm when it operates on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 9.1 shows the decision tree corresponding to the insertion sort algorithm from Section 1.1 operating on an input sequence of three elements.

In a decision tree, each internal node is annotated by $a_i : a_j$ for some i and j in the range $1 \le i, j \le n$, where n is the number of elements in the input sequence. Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (See Section 6.1 for background on permutations.) The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i \le a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i \le a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

A lower bound for the worst case

The length of the longest path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons the sorting algorithm performs. Consequently, the worst-case number of comparisons for a comparison sort corresponds to the height of its decision tree. A lower bound on the heights of decision trees is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 9.1
Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Proof Consider a decision tree of height h that sorts n elements. Since there are $n!$ permutations of n elements, each permutation representing a distinct sorted order, the tree must have at least $n!$ leaves. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \le 2^h ,$$

which, by taking logarithms, implies

$$h \ge \lg(n!) ,$$

since the lg function is monotonically increasing. From Stirling's approximation (2.11), we have

$$n! > \left(\frac{n}{e}\right)^n ,$$

where $e = 2.71828\ldots$ is the base of natural logarithms; thus

$$\begin{aligned} h &\ge \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

■

Corollary 9.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 9.1. ■

Exercises

9.1-1
What is the smallest possible depth of a leaf in a decision tree for a sorting algorithm?

9.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section 3.2.

9.1-3

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

9.1-4

Professor Solomon claims that the $\Omega(n \lg n)$ lower bound for sorting n numbers does not apply to his computer environment, in which the control flow of a program can split three ways after a single comparison $a_i : a_j$, according to whether $a_i < a_j$, $a_i = a_j$, or $a_i > a_j$. Show that the professor is wrong by proving that the number of three-way comparisons required to sort n elements is still $\Omega(n \lg n)$.

9.1-5

Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

9.1-6

You are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint:* It is not rigorous to simply combine the lower bounds for the individual subsequences.)

9.2 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 1 to k, for some integer k. When $k = O(n)$, the sort runs in $O(n)$ time.

The basic idea of counting sort is to determine, for each input element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array. For example, if there are 17 elements less than x, then x belongs in output position 18. This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $\mathit{length}[A] = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[1..k]$ provides temporary working storage.

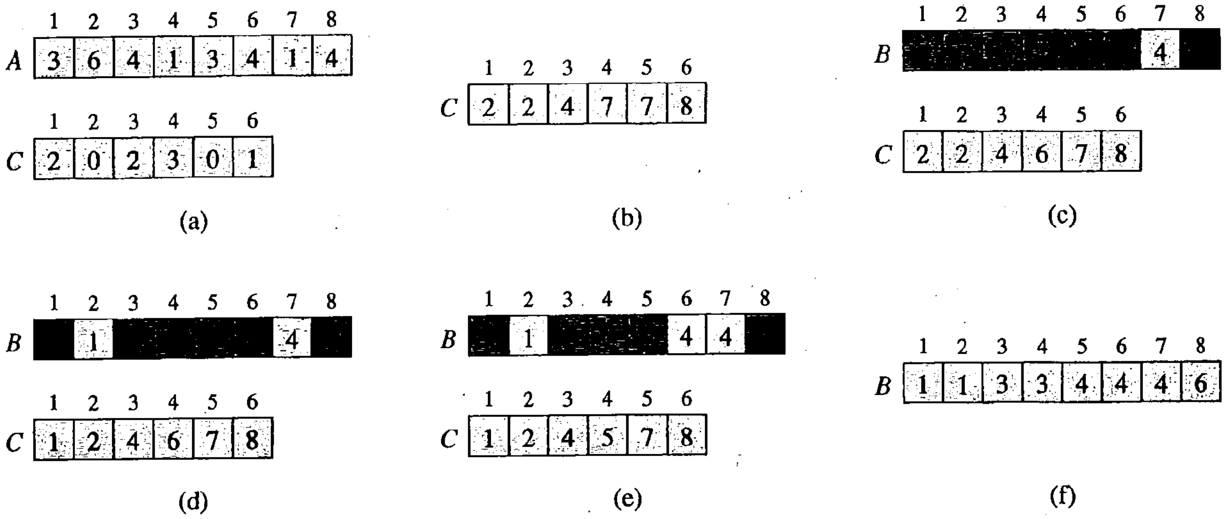


Figure 9.2 The operation of Counting-Sort on an input array $A[1..8]$, where each element of A is a positive integer no larger than $k = 6$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

```
COUNTING-SORT(A, B, k)
 1  for i ← 1 to k
 2      do C[i] ← 0
 3  for j ← 1 to length[A]
 4      do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 2 to k
 7      do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or equal to i.
 9  for j ← length[A] downto 1
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]] − 1
```

Counting sort is illustrated in Figure 9.2. After the initialization in lines 1–2, we inspect each input element in lines 3–4. If the value of an input element is i, we increment $C[i]$. Thus, after lines 3–4, $C[i]$ holds the number of input elements equal to i for each integer $i = 1, 2, \ldots, k$. In lines 6–7, we determine for each $i = 1, 2, \ldots, k$, how many input elements are less than or equal to i; this is done by keeping a running sum of the array C.

Finally, in lines 9–11, we place each element $A[j]$ in its correct sorted position in the output array B. If all n elements are distinct, then when we first enter line 9, for each $A[j]$, the value $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, we decrement $C[A[j]]$ each time we place a value $A[j]$ into the B array; this causes the next input element with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array.

How much time does counting sort require? The **for** loop of lines 1–2 takes time $O(k)$, the **for** loop of lines 3–4 takes time $O(n)$, the **for** loop of lines 6–7 takes time $O(k)$, and the **for** loop of lines 9–11 takes time $O(n)$. Thus, the overall time is $O(k+n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $O(n)$.

Counting sort beats the lower bound of $\Omega(n \lg n)$ proved in Section 9.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison-sort model.

An important property of counting sort is that it is ***stable***: numbers with the same value appear in the output array in the same order as they do in the input array. That is, ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array. Of course, the property of stability is important only when satellite data are carried around with the element being sorted. We shall see why stability is important in the next section.

Exercises

9.2-1

Using Figure 9.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 7,1,3,1,2,4,5,7,2,4,3 \rangle$.

9.2-2

Prove that COUNTING-SORT is stable.

9.2-3

Suppose that the **for** loop in line 9 of the COUNTING-SORT procedure is rewritten:

```
9  for j ← 1 to length[A]
```

Show that the algorithm still works properly. Is the modified algorithm stable?

9.2-4

Suppose that the output of the sorting algorithm is a data stream such as a graphics display. Modify COUNTING-SORT to produce the output in sorted order without using any substantial additional storage besides that in A and C. (*Hint:* Link elements of A that have the same key into lists. Where is a "free" place to keep the pointers for the linked list?)

9.2-5

Describe an algorithm that, given n integers in the range 1 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a\,..\,b]$ in $O(1)$ time. Your algorithm should use $O(n+k)$ preprocessing time.

9.3 Radix sort

Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards are organized into 80 columns, and in each column a hole can be punched in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, only 10 places are used in each column. (The other two places are used for encoding nonnumeric characters.) A d-digit number would then occupy a field of d columns. Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm.

Intuitively, one might want to sort numbers on their *most significant* digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that must be kept track of. (See Exercise 9.3-5.)

Radix-sort solves the problem of card sorting counterintuitively by sorting on the *least significant* digit first. The cards are then combined into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then the entire deck is sorted again on the second least-significant digit and recombined in a like manner. The process continues until the cards have been sorted on all d digits. Remarkably, at that point the cards are fully sorted on the d-digit number. Thus, only d passes through the deck are required to sort. Figure 9.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

It is essential that the digit sorts in this algorithm be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

In a typical computer, which is a sequential random-access machine, radix sort is sometimes used to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a compar-

```
329       720       720       329
457       355       329       355
657       436       436       436
839  ⇒   457  ⇒   839  ⇒   457
436       657       355       657
720       329       457       720
355       839       657       839
            ↑         ↑         ↑
```

Figure 9.3 The operation of radix sort on a list of seven 3-digit numbers. The first column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. The vertical arrows indicate the digit position sorted on to produce each list from the previous one.

ison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
RADIX-SORT($A, d$)
for $i \leftarrow 1$ to $d$
     do use a stable sort to sort array $A$ on digit $i$
```

The correctness of radix sort follows by induction on the column being sorted (see Exercise 9.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 1 to k, and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\Theta(n+k)$. There are d passes, so the total time for radix sort is $\Theta(dn + kd)$. When d is constant and $k = O(n)$, radix sort runs in linear time.

Some computer scientists like to think of the number of bits in a computer word as being $\Theta(\lg n)$. For concreteness, let's say that $d \lg n$ is the number of bits, where d is a positive constant. Then, if each number to be sorted fits in one computer word, we can treat it as a d-digit number in radix-n notation. As a concrete example, consider sorting 1 million 64-bit numbers. By treating these numbers as four-digit, radix-2^{16} numbers, we can sort them in just four passes using radix sort. This compares favorably with a typical $\Theta(n \lg n)$ comparison sort, which requires approximately $\lg n = 20$ operations per number to be sorted. Unfortunately, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$ comparison sorts do. Thus, when primary memory storage is at a premium, an algorithm such as quicksort may be preferable.

Exercises

9.3-1
Using Figure 9.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

9.3-2
Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

9.3-3
Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

9.3-4
Show how to sort n integers in the range 1 to n^2 in $O(n)$ time.

9.3-5 ★
In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

9.4 Bucket sort

Bucket sort runs in linear time on the average. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval $[0, 1)$. (See Section 6.2 for a definition of uniform distribution.)

The idea of bucket sort is to divide the interval $[0, 1)$ into n equal-sized subintervals, or ***buckets***, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over $[0, 1)$, we don't expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Our code for bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \leq A[i] < 1$. The code requires an auxiliary array $B[0\,..\,n-1]$ of linked lists (buckets) and as-

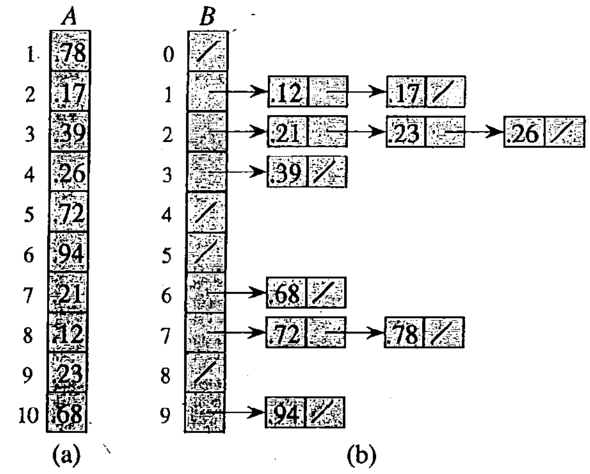


Figure 9.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

sumes that there is a mechanism for maintaining such lists. (Section 11.2 describes how to implement basic operations on linked lists.)

```
BUCKET-SORT($A$)
1  $n \leftarrow length[A]$
2  for $i \leftarrow 1$ to $n$
3       do insert $A[i]$ into list $B[\lfloor \overset{10}{n} A[i] \rfloor]$
4  for $i \leftarrow 0$ to $n - 1$
5       do sort list $B[i]$ with insertion sort
6  concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
```

Figure 9.4 shows the operation of bucket sort on an input array of 10 numbers.

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. If these elements fall in the same bucket, they appear in the proper relative order in the output sequence because their bucket is sorted by insertion sort. Suppose they fall into different buckets, however. Let these buckets be $B[i']$ and $B[j']$, respectively, and assume without loss of generality that $i' < j'$. When the lists of B are concatenated in line 6, elements of bucket $B[i']$ come before elements of $B[j']$, and thus $A[i]$ precedes $A[j]$ in the output sequence. Hence, we must show that $A[i] \leq A[j]$. Assuming the contrary, we have

$$\begin{aligned} i' &= \lfloor nA[i] \rfloor \\ &\geq \lfloor nA[j] \rfloor \\ &= j' , \end{aligned}$$

which is a contradiction, since $i' < j'$. Thus, bucket sort works.

To analyze the running time, observe that all lines except line 5 take $O(n)$ time in the worst case. The total time to examine all buckets in line 5 is $O(n)$, and so the only interesting part of the analysis is the time taken by the insertion sorts in line 5.

To analyze the cost of the insertion sorts, let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 1.2), the expected time to sort the elements in bucket $B[i]$ is $\mathrm{E}\left[O(n_i^2)\right] = O(\mathrm{E}\left[n_i^2\right])$. The total expected time to sort all the elements in all the buckets is therefore

$$\sum_{i=0}^{n-1} O(\mathrm{E}\left[n_i^2\right]) = O\left(\sum_{i=0}^{n-1} \mathrm{E}\left[n_i^2\right]\right) . \tag{9.1}$$

In order to evaluate this summation, we must determine the distribution of each random variable n_i. We have n elements and n buckets. The probability that a given element falls into bucket $B[i]$ is $1/n$, since each bucket is responsible for $1/n$ of the interval $[0, 1)$. Thus, the situation is analogous to the ball-tossing example of Section 6.6.2: we have n balls (elements) and n bins (buckets), and each ball is thrown independently with probability $p = 1/n$ of falling into any particular bucket. Thus, the probability that $n_i = k$ follows the binomial distribution $b(k; n, p)$, which has mean $\mathrm{E}[n_i] = np = 1$ and variance $\mathrm{Var}[n_i] = np(1-p) = 1 - 1/n$. For any random variable X, equation (6.30) gives

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= 1 - \frac{1}{n} + 1^2 \\ &= 2 - \frac{1}{n} \\ &= \Theta(1) . \end{aligned}$$

Using this bound in equation (9.1), we conclude that the expected time for insertion sorting is $O(n)$. Thus, the entire bucket sort algorithm runs in linear expected time.

Exercises

9.4-1

Using Figure 9.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42\rangle$.

9.4-2

What is the worst-case running time for the bucket-sort algorithm? What simple change to the algorithm preserves its linear expected running time and makes its worst-case running time $O(n \lg n)$?

9.4-3 ★

We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \le 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design a $\Theta(n)$ expected-time algorithm to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

9.4-4 ★

A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \le x\}$. Suppose a list of n numbers has a continuous probability distribution function P that is computable in $O(1)$ time. Show how to sort the numbers in linear expected time.

Problems

9-1 Average-case lower bounds on comparison sorting

In this problem, we prove an $\Omega(n \lg n)$ lower bound on the expected running time of any deterministic or randomized comparison sort on n inputs. We begin by examining a deterministic comparison sort A with decision tree T_A. We assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T. Let T be a tree with $k > 1$ leaves, and let RT and LT be the right and left subtrees of T. Show that $D(T) = D(RT) + D(LT) + k$.

c. Let $d(m)$ be the minimum value of $D(T)$ over all trees T with m leaves. Show that $d(k) = \min_{1 \le i \le k} \{d(i) + d(k - i) + k\}$. (*Hint:* Consider a tree T with k leaves that achieves the minimum. Let i be the number of leaves in RT and $k - i$ the number of leaves in LT.)

d. Prove that for a given value of k, the function $i \lg i + (k - i) \lg(k - i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$ for T_A, and conclude that the expected time to sort n elements is $\Omega(n \lg n)$.

Now, consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form RANDOM$(1, r)$ made by algorithm B; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A that makes no more comparisons on the average than B does.

9-2 Sorting in place in linear time

a. Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. Give a simple, linear-time algorithm for sorting the n data records in place. Use no storage of more than constant size in addition to the storage provided by the array.

b. Can your sort from part (a) be used to radix sort n records with b-bit keys in $O(bn)$ time? Explain how or why not.

c. Suppose that the n records havc keys in the range from 1 to k. Show how to modify counting sort so that the records can be sorted in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. (*Hint:* How would you do it for $k = 3$?)

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [72]. Knuth's comprehensive treatise on sorting [123] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Lower bounds for sorting using generalizations of the decision-tree model were studied comprehensively by Ben-Or [23].

Knuth credits H. H. Seward with inventing counting sort in 1954, and also with the idea of combining counting sort with radix sort. Radix sorting by the least-significant digit first appears to be a folk algorithm widely used by operators of mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by E. J. Isaac and R. C. Singleton.

10 Medians and Order Statistics

The ith ***order statistic*** of a set of n elements is the ith smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the nth order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = (n + 1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n+1)/2 \rfloor$ and $i = \lceil (n+1)/2 \rceil$.

This chapter addresses the problem of selecting the ith order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. The ***selection problem*** can be specified formally as follows:

Input: A set A of n (distinct) numbers and a number i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

The selection problem can be solved in $O(n \lg n)$ time, since we can sort the numbers using heapsort or merge sort and then simply index the ith element in the output array. There are faster algorithms, however.

In Section 10.1, we examine the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which is investigated in the subsequent two sections. Section 10.2 analyzes a practical algorithm that achieves an $O(n)$ bound on the running time in the average case. Section 10.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

10.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? We can easily obtain an upper bound of $n - 1$ comparisons: examine each element of the set in turn and keep track of the smallest element seen so far. In the following procedure, we assume that the set resides in array A, where $length[A] = n$.

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A]
3       do if min > A[i]
4             then min ← A[i]
5  return min
```

Finding the maximum can, of course, be accomplished with $n - 1$ comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. The key observation is that every element except the winner must lose at least one match. Hence, $n - 1$ comparisons are necessary to determine the minimum, and the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

An interesting fine point of the analysis is the determination of the expected number of times that line 4 is executed. Problem 6-2 asks you to show that this expectation is $\Theta(\lg n)$.

Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum of each coordinate.

It is not too difficult to devise an algorithm that can find both the minimum and the maximum of n elements using the asymptotically optimal $\Omega(n)$ number of comparisons. Simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.

In fact, only $3\lceil n/2 \rceil$ comparisons are necessary to find both the minimum and the maximum. To do this, we maintain the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, however, at a cost of two comparisons per element, we process elements in pairs. We compare pairs of elements from the input first with *each other*, and then compare the smaller to the current minimum and the larger to the current maximum, at a cost of three comparisons for every two elements.

Exercises

10.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

10.1-2 ★

Show that $\lceil 3n/2 \rceil - 2$ comparisons are necessary in the worst case to find both the maximum and minimum of n numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

10.2 Selection in expected linear time

The general selection problem appears more difficult than the simple problem of finding a minimum, yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. In this section, we present a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 8. As in quicksort, the idea is to partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT only works on one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected time of RANDOMIZED-SELECT is $\Theta(n)$.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 8.3. Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for RANDOMIZED-SELECT returns the ith smallest element of the array $A[p\,..\,r]$.

```
RANDOMIZED-SELECT($A, p, r, i$)
1  if $p = r$
2     then return $A[p]$
3  $q \leftarrow$ RANDOMIZED-PARTITION($A, p, r$)
4  $k \leftarrow q - p + 1$
5  if $i \le k$
6     then return RANDOMIZED-SELECT($A, p, q, i$)
7     else return RANDOMIZED-SELECT($A, q + 1, r, i - k$)
```

After RANDOMIZED-PARTITION is executed in line 3 of the algorithm, the array $A[p\,..\,r]$ is partitioned into two nonempty subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ such that each element of $A[p\,..\,q]$ is less than each element of $A[q+1\,..\,r]$. Line 4 of the algorithm computes the number k of elements in the subarray $A[p\,..\,q]$. The algorithm now determines in which of the two subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ the ith smallest element lies. If $i \leq k$, then the desired element lies on the low side of the partition, and it is recursively selected from the subarray in line 6. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p\,..\,r]$—namely, the elements of $A[p\,..\,q]$—the desired element is the $(i-k)$th smallest element of $A[q+1\,..\,r]$, which is found recursively in line 7.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element. The algorithm works well in the average case, though, and because it is randomized, no particular input elicits the worst-case behavior.

We can obtain an upper bound $T(n)$ on the expected time required by RANDOMIZED-SELECT on an input array of n elements as follows. We observed in Section 8.4 that the algorithm RANDOMIZED-PARTITION produces a partition whose low side has 1 element with probability $2/n$ and i elements with probability $1/n$ for $i = 2, 3, \ldots, n-1$. Assuming that $T(n)$ is monotonically increasing, in the worst case RANDOMIZED-SELECT is always unlucky in that the ith element is determined to be on the larger side of the partition. Thus, we get the recurrence

$$\begin{aligned}
T(n) &\leq \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\
&\leq \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2\rceil}^{n-1} T(k)\right) + O(n) \\
&= \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} T(k) + O(n)\,.
\end{aligned}$$

The second line follows from the first since $\max(1, n-1) = n-1$ and

$$\max(k, n-k) = \begin{cases} k & \text{if } k \geq \lceil n/2\rceil\,, \\ n-k & \text{if } k < \lceil n/2\rceil\,. \end{cases}$$

If n is odd, each term $T(\lceil n/2\rceil), T(\lceil n/2\rceil + 1), \ldots, T(n-1)$ appears twice in the summation, and if n is even, each term $T(\lceil n/2\rceil + 1), T(\lceil n/2\rceil + 2), \ldots, T(n-1)$ appears twice and the term $T(\lceil n/2\rceil)$ appears once. In either case, the summation of the first line is bounded from above by the summation of the second line. The third line follows from the second since in the worst case $T(n-1) = O(n^2)$, and thus the term $\frac{1}{n}T(n-1)$ can be absorbed by the term $O(n)$.

We solve the recurrence by substitution. Assume that $T(n) \leq cn$ for some constant c that satisfies the initial conditions of the recurrence. Using this inductive hypothesis, we have

$$\begin{aligned}
T(n) &\le \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} ck + O(n) \\
&\le \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil-1} k\right) + O(n) \\
&= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)\left\lceil\frac{n}{2}\right\rceil\right) + O(n) \\
&\le c(n-1) - \frac{c}{n}\left(\frac{n}{2} - 1\right)\left(\frac{n}{2}\right) + O(n) \\
&= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n) \\
&\le cn\ ,
\end{aligned}$$

since we can pick c large enough so that $c(n/4 + 1/2)$ dominates the $O(n)$ term.

Thus, any order statistic, and in particular the median, can be determined on average in linear time.

Exercises

10.2-1

Write an iterative version of RANDOMIZED-SELECT.

10.2-2

Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1\rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

10.2-3

Recall that in the presence of equal elements, the RANDOMIZED-PARTITION procedure partitions the subarray $A[p\,..\,r]$ into two nonempty subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ such that each element in $A[p\,..\,q]$ is less than *or equal to* every element in $A[q+1\,..\,r]$. If equal elements are present, does the RANDOMIZED-SELECT procedure work correctly?

10.3 Selection in worst-case linear time

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. The idea behind the algorithm, however, is to *guarantee* a good split when the array is partitioned. SELECT uses the deterministic partitioning algorithm PARTITION

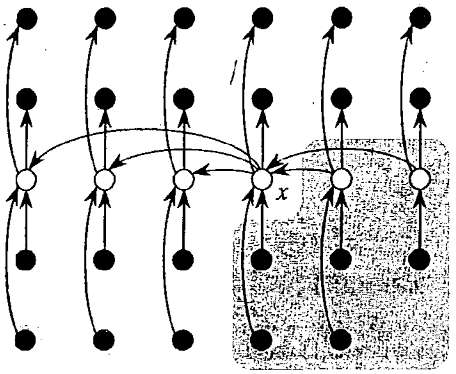


Figure 10.1 Analysis of the algorithm SELECT. The n elements are represented by small circles, and each group occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. Arrows are drawn from larger elements to smaller, from which it can be seen that 3 out of every group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements greater than x are shown on a shaded background.

from quicksort (see Section 8.1), modified to take the element to partition around as an input parameter.

The SELECT algorithm determines the ith smallest of an input array of n elements by executing the following steps.

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by insertion sorting the elements of each group (of which there are 5 at most) and taking its middle element. (If the group has an even number of elements, take the larger of the two medians.)
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2.
4. Partition the input array around the median-of-medians x using a modified version of PARTITION. Let k be the number of elements on the low side of the partition, so that $n - k$ is the number of elements on the high side.
5. Use SELECT recursively to find the ith smallest element on the low side if $i \leq k$, or the $(i - k)$th smallest element on the high side if $i > k$.

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element x. Figure 10.1 is helpful in visualizing this bookkeeping. At least half of the medians found in step 2 are greater than or equal to the median-of-medians x. Thus, at least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x

itself. Discounting these two groups, it follows that the number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

Similarly, the number of elements that are less than x is at least $3n/10-6$. Thus, in the worst case, SELECT is called recursively on at most $7n/10+6$ elements in step 5.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5\rceil)$, and step 5 takes time at most $T(7n/10+6)$, assuming that T is monotonically increasing. Note that $7n/10+6 < n$ for $n > 20$ and that any input of 80 or fewer elements requires $O(1)$ time. We can therefore obtain the recurrence

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 80 , \\ T(\lceil n/5\rceil) + T(7n/10+6) + O(n) & \text{if } n > 80 . \end{cases}$$

We show that the running time is linear by substitution. Assume that $T(n) \leq cn$ for some constant c and all $n \leq 80$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\leq c\lceil n/5\rceil + c(7n/10+6) + O(n) \\ &\leq cn/5 + c + 7cn/10 + 6c + O(n) \\ &\leq 9cn/10 + 7c + O(n) \\ &\leq cn , \end{aligned}$$

since we can pick c large enough so that $c(n/10-7)$ is larger than the function described by the $O(n)$ term for all $n > 80$. The worst-case running time of SELECT is therefore linear.

As in a comparison sort (see Section 9.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Thus, the linear-time behavior is not a result of assumptions about the input, as was the case for the sorting algorithms in Chapter 9. Sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 9-1), and thus the method of sorting and indexing presented in the introduction to this chapter is asymptotically inefficient.

Exercises

10.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? How about groups of 3?

10.3-2

Analyze SELECT to show that the number of elements greater than the median-of-medians x and the number of elements less than x is at least $\lceil n/4 \rceil$ if $n \geq 38$.

10.3-3

Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case.

10.3-4 ⋆

Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i-1$ smaller elements and the $n-i$ larger elements without performing any additional comparisons.

10.3-5

Given a "black-box" worst-case linear-time median subroutine, give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

10.3-6

The kth ***quantiles*** of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

10.3-7

Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

10.3-8

Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

10.3-9

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. From each well, a spur pipeline is to be connected directly to the main pipeline along a shortest path (either north or south), as shown in Figure 10.2. Given x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline (the one that minimizes the total length of the spurs)? Show that the optimal location can be determined in linear time.

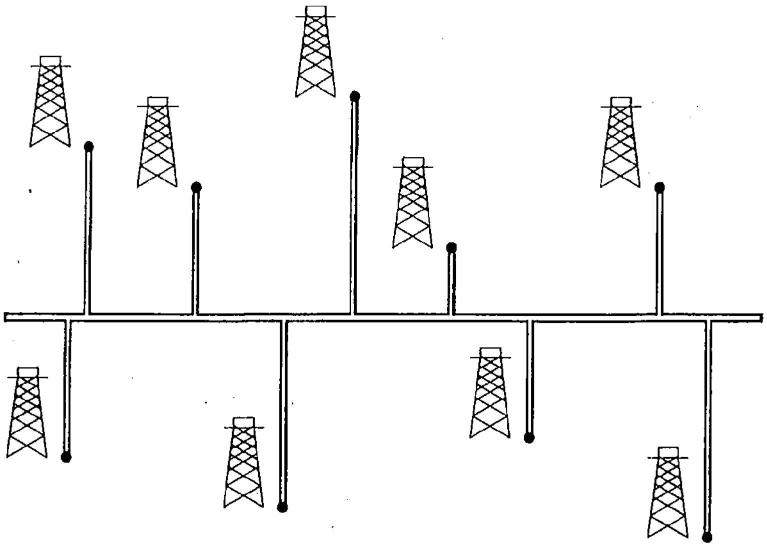


Figure 10.2 We want to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

Problems

10-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the methods in terms of n and i.

a. Sort the numbers and list the i largest.

b. Build a priority queue from the numbers and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition, and sort the i largest numbers.

10-2 Weighted median

For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the ***weighted median*** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i \le \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} \, .$$

a. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

b. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

c. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 10.3.

The ***post-office location problem*** is defined as follows. We are given n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i \, d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

e. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance***: $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

10-3 Small order statistics

The worst-case number $T(n)$ of comparisons used by SELECT to select the ith order statistic from n numbers was shown to satisfy $T(n) = \Theta(n)$, but the constant hidden by the Θ-notation is rather large. When i is small relative to n, we can implement a different procedure that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where $i \le n/2$ and

$$U_i(n) = \begin{cases} T(n) & \text{if } n \le 2i \, , \\ n/2 + U_i(n/2) + T(2i) & \text{otherwise} \, . \end{cases}$$

(*Hint:* Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

b. Show that $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Show that if i is a constant, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \ge 2$, then $U_i(n) = n + O(T(2n/k) \lg k)$.

Chapter notes

The worst-case median-finding algorithm was invented by Blum, Floyd, Pratt, Rivest, and Tarjan [29]. The fast average-time version is due to Hoare [97]. Floyd and Rivest [70] have developed an improved average-time version that partitions around an element recursively selected from a small sample of the elements.

EXHIBIT B

Data Clustering: A Review

A.K. JAIN

Michigan State University

M.N. MURTY

Indian Institute of Science

AND

P.J. FLYNN

The Ohio State University

Clustering is the unsupervised classification of patterns (observations, data items, or feature vectors) into groups (clusters). The clustering problem has been addressed in many contexts and by researchers in many disciplines; this reflects its broad appeal and usefulness as one of the steps in exploratory data analysis. However, clustering is a difficult problem combinatorially, and differences in assumptions and contexts in different communities has made the transfer of useful generic concepts and methodologies slow to occur. This paper presents an overview of pattern clustering methods from a statistical pattern recognition perspective, with a goal of providing useful advice and references to fundamental concepts accessible to the broad community of clustering practitioners. We present a taxonomy of clustering techniques, and identify cross-cutting themes and recent advances. We also describe some important applications of clustering algorithms such as image segmentation, object recognition, and information retrieval.

Categories and Subject Descriptors: I.5.1 [**Pattern Recognition**]: Models; I.5.3 [**Pattern Recognition**]: Clustering; I.5.4 [**Pattern Recognition**]: Applications—*Computer vision*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering*; I.2.6 [**Artificial Intelligence**]: Learning—*Knowledge acquisition*

General Terms: Algorithms

Additional Key Words and Phrases: Cluster analysis, clustering applications, exploratory data analysis, incremental clustering, similarity indices, unsupervised learning

Section 6.1 is based on the chapter "Image Segmentation Using Clustering" by A.K. Jain and P.J. Flynn, *Advances in Image Understanding: A Festschrift for Azriel Rosenfeld* (K. Bowyer and N. Ahuja, Eds.), 1996 IEEE Computer Society Press, and is used by permission of the IEEE Computer Society.

Authors' addresses: A. Jain, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, MI 48824; M. Murty, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India; P. Flynn, Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210.

CONTENTS

1. INTRODUCTION

1.1 Motivation

Data analysis underlies many computing applications, either in a design phase or as part of their on-line operations. Data analysis procedures can be dichotomized as either exploratory or confirmatory, based on the availability of appropriate models for the data source, but a key element in both types of procedures (whether for hypothesis formation or decision-making) is the grouping, or classification of measurements based on either (i) goodness-of-fit to a postulated model, or (ii) natural groupings (clustering) revealed through analysis. Cluster analysis is the organization of a collection of patterns (usually represented as a vector of measurements, or a point in a multidimensional space) into clusters based on similarity. Intuitively, patterns within a valid cluster are more similar to each other than they are to a pattern belonging to a different cluster. An example of clustering is depicted in Figure 1. The input patterns are shown in Figure 1(a), and the desired clusters are shown in Figure 1(b). Here, points belonging to the same cluster are given the same label. The variety of techniques for representing data, measuring proximity (similarity) between data elements, and grouping data elements has produced a rich and often confusing assortment of clustering methods.

It is important to understand the difference between clustering (unsupervised classification) and discriminant analysis (supervised classification). In supervised classification, we are provided with a collection of *labeled* (preclassified) patterns; the problem is to label a newly encountered, yet unlabeled, pattern. Typically, the given labeled (*training*) patterns are used to learn the descriptions of classes which in turn are used to label a new pattern. In the case of clustering, the problem is to group a given collection of unlabeled patterns into meaningful clusters. In a sense, labels are associated with clusters also, but these category labels are *data driven*; that is, they are obtained solely from the data.

Clustering is useful in several exploratory pattern-analysis, grouping, decision-making, and machine-learning situations, including data mining, document retrieval, image segmentation, and pattern classification. However, in many such problems, there is little prior information (e.g., statistical models) available about the data, and the decision-maker must make as few assumptions about the data as possible. It is under these restrictions that clustering methodology is particularly appropriate for the exploration of interrelationships among the data points to make an assessment (perhaps preliminary) of their structure.

The term "clustering" is used in several research communities to describe

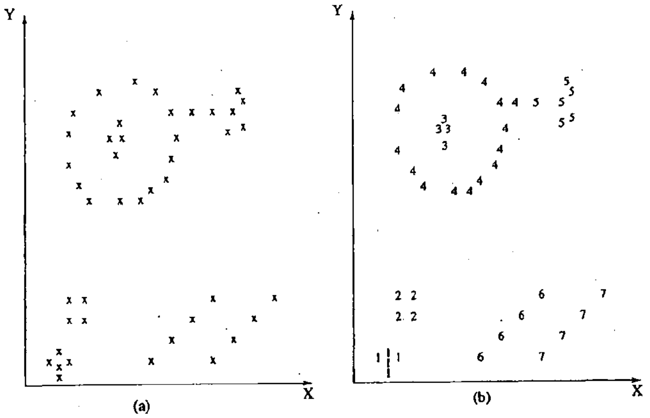


Figure 1. Data clustering.

methods for grouping of unlabeled data. These communities have different terminologies and assumptions for the components of the clustering process and the contexts in which clustering is used. Thus, we face a dilemma regarding the scope of this survey. The production of a truly comprehensive survey would be a monumental task given the sheer mass of literature in this area. The accessibility of the survey might also be questionable given the need to reconcile very different vocabularies and assumptions regarding clustering in the various communities.

The goal of this paper is to survey the core concepts and techniques in the large subset of cluster analysis with its roots in statistics and decision theory. Where appropriate, references will be made to key concepts and techniques arising from clustering methodology in the machine-learning and other communities.

The audience for this paper includes practitioners in the pattern recognition and image analysis communities (who should view it as a summarization of current practice), practitioners in the machine-learning communities (who should view it as a snapshot of a closely related field with a rich history of well-understood techniques), and the broader audience of scientific professionals (who should view it as an accessible introduction to a mature field that is making important contributions to computing application areas).

1.2 Components of a Clustering Task

Typical pattern clustering activity involves the following steps [Jain and Dubes 1988]:

(1) pattern representation (optionally including feature extraction and/or selection),

(2) definition of a pattern proximity measure appropriate to the data domain,

(3) clustering or grouping,

(4) data abstraction (if needed), and

(5) assessment of output (if needed).

Figure 2 depicts a typical sequencing of the first three of these steps, including a feedback path where the grouping process output could affect subsequent feature extraction and similarity computations.

Pattern representation refers to the number of classes, the number of available patterns, and the number, type, and scale of the features available to the clustering algorithm. Some of this information may not be controllable by the

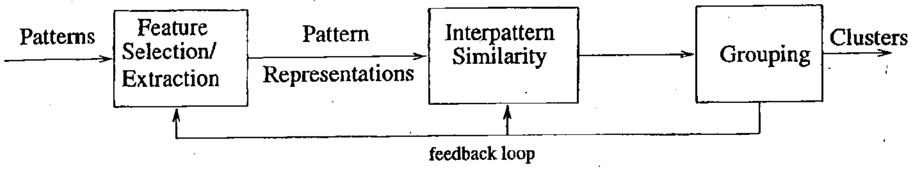


Figure 2. Stages in clustering.

practitioner. *Feature selection* is the process of identifying the most effective subset of the original features to use in clustering. *Feature extraction* is the use of one or more transformations of the input features to produce new salient features. Either or both of these techniques can be used to obtain an appropriate set of features to use in clustering.

Pattern proximity is usually measured by a distance function defined on pairs of patterns. A variety of distance measures are in use in the various communities [Anderberg 1973; Jain and Dubes 1988; Diday and Simon 1976]. A simple distance measure like Euclidean distance can often be used to reflect dissimilarity between two patterns, whereas other similarity measures can be used to characterize the conceptual similarity between patterns [Michalski and Stepp 1983]. Distance measures are discussed in Section 4.

The *grouping* step can be performed in a number of ways. The output clustering (or clusterings) can be hard (a partition of the data into groups) or fuzzy (where each pattern has a variable degree of membership in each of the output clusters). Hierarchical clustering algorithms produce a nested series of partitions based on a criterion for merging or splitting clusters based on similarity. Partitional clustering algorithms identify the partition that optimizes (usually locally) a clustering criterion. Additional techniques for the grouping operation include probabilistic [Brailovski 1991] and graph-theoretic [Zahn 1971] clustering methods. The variety of techniques for cluster formation is described in Section 5.

Data abstraction is the process of extracting a simple and compact representation of a data set. Here, simplicity is either from the perspective of automatic analysis (so that a machine can perform further processing efficiently) or it is human-oriented (so that the representation obtained is easy to comprehend and intuitively appealing). In the clustering context, a typical data abstraction is a compact description of each cluster, usually in terms of cluster prototypes or representative patterns such as the centroid [Diday and Simon 1976].

How is the output of a clustering algorithm evaluated? What characterizes a 'good' clustering result and a 'poor' one? All clustering algorithms will, when presented with data, produce clusters — regardless of whether the data contain clusters or not. If the data does contain clusters, some clustering algorithms may obtain 'better' clusters than others. The assessment of a clustering procedure's output, then, has several facets. One is actually an assessment of the data domain rather than the clustering algorithm itself— data which do not contain clusters should not be processed by a clustering algorithm. The study of *cluster tendency*, wherein the input data are examined to see if there is any merit to a cluster analysis prior to one being performed, is a relatively inactive research area, and will not be considered further in this survey. The interested reader is referred to Dubes [1987] and Cheng [1995] for information.

Cluster validity analysis, by contrast, is the assessment of a clustering procedure's output. Often this analysis uses a specific criterion of optimality; however, these criteria are usually arrived at subjectively. Hence, little in the way of 'gold standards' exist in clustering except in well-prescribed subdomains. Validity assessments are objective [Dubes 1993] and are performed to determine whether the output is meaningful. A clustering structure is valid if it cannot reasonably have occurred by chance or as an artifact of a clustering algorithm. When statistical approaches to clustering are used, validation is accomplished by carefully applying statistical methods and testing hypotheses. There are three types of validation studies. An *external* assessment of validity compares the recovered structure to an *a priori* structure. An *internal* examination of validity tries to determine if the structure is intrinsically appropriate for the data. A *relative* test compares two structures and measures their relative merit. Indices used for this comparison are discussed in detail in Jain and Dubes [1988] and Dubes [1993], and are not discussed further in this paper.

1.3 The User's Dilemma and the Role of Expertise

The availability of such a vast collection of clustering algorithms in the literature can easily confound a user attempting to select an algorithm suitable for the problem at hand. In Dubes and Jain [1976], a set of admissibility criteria defined by Fisher and Van Ness [1971] are used to compare clustering algorithms. These admissibility criteria are based on: (1) the manner in which clusters are formed, (2) the structure of the data, and (3) sensitivity of the clustering technique to changes that do not affect the structure of the data. However, there is no critical analysis of clustering algorithms dealing with the important questions such as

—How should the data be normalized?

—Which similarity measure is appropriate to use in a given situation?

—How should domain knowledge be utilized in a particular clustering problem?

—How can a vary large data set (say, a million patterns) be clustered efficiently?

These issues have motivated this survey, and its aim is to provide a perspective on the state of the art in clustering methodology and algorithms. With such a perspective, an informed practitioner should be able to confidently assess the tradeoffs of different techniques, and ultimately make a competent decision on a technique or suite of techniques to employ in a particular application.

There is no clustering technique that is universally applicable in uncovering the variety of structures present in multidimensional data sets. For example, consider the two-dimensional data set shown in Figure 1(a). Not all clustering techniques can uncover all the clusters present here with equal facility, because clustering algorithms often contain implicit assumptions about cluster shape or multiple-cluster configurations based on the similarity measures and grouping criteria used.

Humans perform competitively with automatic clustering procedures in two dimensions, but most real problems involve clustering in higher dimensions. It is difficult for humans to obtain an intuitive interpretation of data embedded in a high-dimensional space. In addition, data hardly follow the "ideal" structures (e.g., hyperspherical, linear) shown in Figure 1. This explains the large number of clustering algorithms which continue to appear in the literature; each new clustering algorithm performs slightly better than the existing ones on a specific distribution of patterns.

It is essential for the user of a clustering algorithm to not only have a thorough understanding of the particular technique being utilized, but also to know the details of the data gathering process and to have some domain expertise; the more information the user has about the data at hand, the more likely the user would be able to succeed in assessing its true class structure [Jain and Dubes 1988]. This domain information can also be used to improve the quality of feature extraction, similarity computation, grouping, and cluster representation [Murty and Jain 1995].

Appropriate constraints on the data source can be incorporated into a clustering procedure. One example of this is *mixture resolving* [Titterington et al. 1985], wherein it is assumed that the data are drawn from a mixture of an unknown number of densities (often assumed to be multivariate Gaussian). The clustering problem here is to identify the number of mixture components and the parameters of each component. The concept of *density* clustering and a methodology for decomposition of feature spaces [Bajcsy 1997] have also been incorporated into traditional clustering methodology, yielding a technique for extracting overlapping clusters.

1.4 History

Even though there is an increasing interest in the use of clustering methods in pattern recognition [Anderberg 1973], image processing [Jain and Flynn 1996] and information retrieval [Rasmussen 1992; Salton 1991], clustering has a rich history in other disciplines [Jain and Dubes 1988] such as biology, psychiatry, psychology, archaeology, geology, geography, and marketing. Other terms more or less synonymous with clustering include *unsupervised learning* [Jain and Dubes 1988], *numerical taxonomy* [Sneath and Sokal 1973], *vector quantization* [Oehler and Gray 1995], and *learning by observation* [Michalski and Stepp 1983]. The field of spatial analysis of point patterns [Ripley 1988] is also related to cluster analysis. The importance and interdisciplinary nature of clustering is evident through its vast literature.

A number of books on clustering have been published [Jain and Dubes 1988; Anderberg 1973; Hartigan 1975; Spath 1980; Duran and Odell 1974; Everitt 1993; Backer 1995], in addition to some useful and influential review papers. A survey of the state of the art in clustering *circa* 1978 was reported in Dubes and Jain [1980]. A comparison of various clustering algorithms for constructing the minimal spanning tree and the short spanning path was given in Lee [1981]. Cluster analysis was also surveyed in Jain et al. [1986]. A review of image segmentation by clustering was reported in Jain and Flynn [1996]. Comparisons of various combinatorial optimization schemes, based on experiments, have been reported in Mishra and Raghavan [1994] and Al-Sultan and Khan [1996].

1.5 Outline

This paper is organized as follows. Section 2 presents definitions of terms to be used throughout the paper. Section 3 summarizes pattern representation, feature extraction, and feature selection. Various approaches to the computation of proximity between patterns are discussed in Section 4. Section 5 presents a taxonomy of clustering approaches, describes the major techniques in use, and discusses emerging techniques for clustering incorporating non-numeric constraints and the clustering of large sets of patterns. Section 6 discusses applications of clustering methods to image analysis and data mining problems. Finally, Section 7 presents some concluding remarks.

2. DEFINITIONS AND NOTATION

The following terms and notation are used throughout this paper.

—A *pattern* (or *feature vector, observation*, or *datum*) $\mathbf{x}$ is a single data item used by the clustering algorithm. It typically consists of a vector of d measurements: $\mathbf{x} = (x_1, \ldots x_d)$.

—The individual scalar components x_i of a pattern $\mathbf{x}$ are called *features* (or *attributes*).

—d is the *dimensionality* of the pattern or of the pattern space.

—A *pattern set* is denoted $\mathcal{X} = \{\mathbf{x}_1, \ldots \mathbf{x}_n\}$. The ith pattern in $\mathcal{X}$ is denoted $\mathbf{x}_i = (x_{i,1}, \ldots x_{i,d})$. In many cases a pattern set to be clustered is viewed as an $n \times d$ *pattern matrix.*

—A *class,* in the abstract, refers to a state of nature that governs the pattern generation process in some cases. More concretely, a class can be viewed as a source of patterns whose distribution in feature space is governed by a probability density specific to the class. Clustering techniques attempt to group patterns so that the classes thereby obtained reflect the different pattern generation processes represented in the pattern set.

—*Hard* clustering techniques assign a *class label* l_i to each patterns $\mathbf{x}_i$, identifying its class. The set of all labels for a pattern set $\mathcal{X}$ is $\mathcal{L} = \{l_1, \ldots l_n\}$, with $l_i \in \{1, \cdots, k\}$, where k is the number of clusters.

—*Fuzzy* clustering procedures assign to each input pattern $\mathbf{x}_i$ a fractional degree of membership f_{ij} in each output cluster j.

—A *distance measure* (a specialization of a proximity measure) is a metric (or quasi-metric) on the feature space used to quantify the similarity of patterns.

3. PATTERN REPRESENTATION, FEATURE SELECTION AND EXTRACTION

There are no theoretical guidelines that suggest the appropriate patterns and features to use in a specific situation. Indeed, the pattern generation process is often not directly controllable; the user's role in the pattern representation process is to gather facts and conjectures about the data, optionally perform feature selection and extraction, and design the subsequent elements of the clustering system. Because of the difficulties surrounding pattern representation, it is conveniently assumed that the pattern representation is available prior to clustering. Nonetheless, a careful investigation of the available features and any available transformations (even simple ones) can yield significantly improved clustering results. A good pattern representation can often yield a simple and easily understood clustering; a poor pattern representation may yield a complex clustering whose true structure is difficult or impossible to discern. Figure 3 shows a simple example. The points in this 2D feature space are arranged in a curvilinear cluster of approximately constant distance from the origin. If one chooses Cartesian coordinates to represent the patterns, many clustering algorithms would be likely to fragment the cluster into two or more clusters, since it is not compact. If, however, one uses a polar coordinate representation for the clusters, the radius coordinate exhibits tight clustering and a one-cluster solution is likely to be easily obtained.

A pattern can measure either a physical object (e.g., a chair) or an abstract notion (e.g., a style of writing). As noted above, patterns are represented conventionally as multidimensional vectors, where each dimension is a single feature [Duda and Hart 1973]. These features can be either quantitative or qualitative. For example, if *weight* and *color* are the two features used, then (20, *black*) is the representation of a black object with 20 units of weight. The features can be subdivided into the following types [Gowda and Diday 1992]:

(1) Quantitative features: e.g.
   (a) continuous values (e.g., weight);
   (b) discrete values (e.g., the number of computers);
   (c) interval values (e.g., the duration of an event).

(2) Qualitative features:
   (a) nominal or unordered (e.g., color);

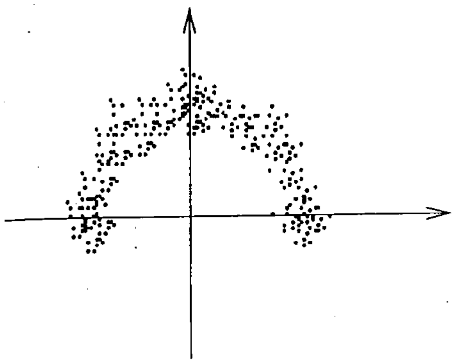


Figure 3. A curvilinear cluster whose points are approximately equidistant from the origin. Different pattern representations (coordinate systems) would cause clustering algorithms to yield different results for this data (see text).

(b) ordinal (e.g., military rank or qualitative evaluations of temperature ("cool" or "hot") or sound intensity ("quiet" or "loud")).

Quantitative features can be measured on a ratio scale (with a meaningful reference value, such as temperature), or on nominal or ordinal scales.

One can also use structured features [Michalski and Stepp 1983] which are represented as trees, where the parent node represents a generalization of its child nodes. For example, a parent node "vehicle" may be a generalization of children labeled "cars," "buses," "trucks," and "motorcycles." Further, the node "cars" could be a generalization of cars of the type "Toyota," "Ford," "Benz," etc. A generalized representation of patterns, called *symbolic objects* was proposed in Diday [1988]. Symbolic objects are defined by a logical conjunction of events. These events link values and features in which the features can take one or more values and all the objects need not be defined on the same set of features.

It is often valuable to isolate only the most descriptive and discriminatory features in the input set, and utilize those features exclusively in subsequent analysis. Feature selection techniques identify a subset of the existing features for subsequent use, while feature extraction techniques compute new features from the original set. In either case, the goal is to improve classification performance and/or computational efficiency. Feature selection is a well-explored topic in statistical pattern recognition [Duda and Hart 1973]; however, in a clustering context (i.e., lacking class labels for patterns), the feature selection process is of necessity ad hoc, and might involve a trial-and-error process where various subsets of features are selected, the resulting patterns clustered, and the output evaluated using a validity index. In contrast, some of the popular feature extraction processes (e.g., principal components analysis [Fukunaga 1990]) do not depend on labeled data and can be used directly. Reduction of the number of features has an additional benefit, namely the ability to produce output that can be visually inspected by a human.

4. SIMILARITY MEASURES

Since similarity is fundamental to the definition of a cluster, a measure of the similarity between two patterns drawn from the same feature space is essential to most clustering procedures. Because of the variety of feature types and scales, the distance measure (or measures) must be chosen carefully. It is most common to calculate the *dissimilarity* between two patterns using a distance measure defined on the feature space. We will focus on the well-known distance measures used for patterns whose features are all continuous.

The most popular metric for continuous features is the *Euclidean distance*

$$d_2(\mathbf{x}_i, \mathbf{x}_j) = (\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2)^{1/2}$$

$$= \|\mathbf{x}_i - \mathbf{x}_j\|_2,$$

which is a special case ($p=2$) of the Minkowski metric

$$d_p(\mathbf{x}_i, \mathbf{x}_j) = (\sum_{k=1}^{d} |x_{i,k} - x_{j,k}|^p)^{1/p}$$

$$= \|\mathbf{x}_i - \mathbf{x}_j\|_p.$$

The Euclidean distance has an intuitive appeal as it is commonly used to evaluate the proximity of objects in two or three-dimensional space. It works well when a data set has "compact" or "isolated" clusters [Mao and Jain 1996]. The drawback to direct use of the Minkowski metrics is the tendency of the largest-scaled feature to dominate the others. Solutions to this problem include normalization of the continuous features (to a common range or variance) or other weighting schemes. Linear correlation among features can also distort distance measures; this distortion can be alleviated by applying a whitening transformation to the data or by using the squared Mahalanobis distance

$$d_M(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)\Sigma^{-1}(\mathbf{x}_i - \mathbf{x}_j)^T,$$

where the patterns $\mathbf{x}_i$ and $\mathbf{x}_j$ are assumed to be row vectors, and Σ is the sample covariance matrix of the patterns or the known covariance matrix of the pattern generation process; $d_M(\cdot\ ,\cdot)$ assigns different weights to different features based on their variances and pairwise linear correlations. Here, it is implicitly assumed that class conditional densities are unimodal and characterized by multidimensional spread, i.e., that the densities are multivariate Gaussian. The regularized Mahalanobis distance was used in Mao and Jain [1996] to extract hyperellipsoidal clusters. Recently, several researchers [Huttenlocher et al. 1993; Dubuisson and Jain 1994] have used the Hausdorff distance in a point set matching context.

Some clustering algorithms work on a matrix of proximity values instead of on the original pattern set. It is useful in such situations to precompute all the $n(n-1)/2$ pairwise distance values for the n patterns and store them in a (symmetric) matrix.

Computation of distances between patterns with some or all features being noncontinuous is problematic, since the different types of features are not comparable and (as an extreme example) the notion of proximity is effectively binary-valued for nominal-scaled features. Nonetheless, practitioners (especially those in machine learning, where mixed-type patterns are common) have developed proximity measures for heterogeneous type patterns. A recent example is Wilson and Martinez [1997], which proposes a combination of a modified Minkowski metric for continuous features and a distance based on counts (population) for nominal attributes. A variety of other metrics have been reported in Diday and Simon [1976] and Ichino and Yaguchi [1994] for computing the similarity between patterns represented using quantitative as well as qualitative features.

Patterns can also be represented using string or tree structures [Knuth 1973]. Strings are used in syntactic clustering [Fu and Lu 1977]. Several measures of similarity between strings are described in Baeza-Yates [1992]. A good summary of similarity measures between trees is given by Zhang [1995]. A comparison of syntactic and statistical approaches for pattern recognition using several criteria was presented in Tanaka [1995] and the conclusion was that syntactic methods are inferior in every aspect. Therefore, we do not consider syntactic methods further in this paper.

There are some distance measures reported in the literature [Gowda and Krishna 1977; Jarvis and Patrick 1973] that take into account the effect of surrounding or neighboring points. These surrounding points are called *context* in Michalski and Stepp [1983]. The similarity between two points $\mathbf{x}_i$ and $\mathbf{x}_j$, given this context, is given by

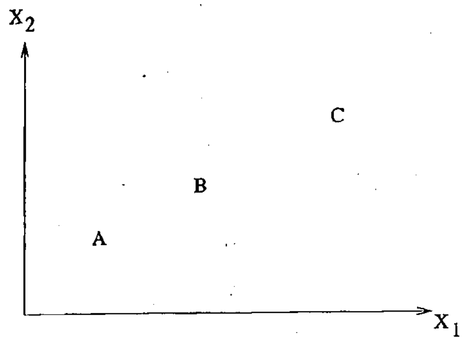


Figure 4. A and B are more similar than A and C.

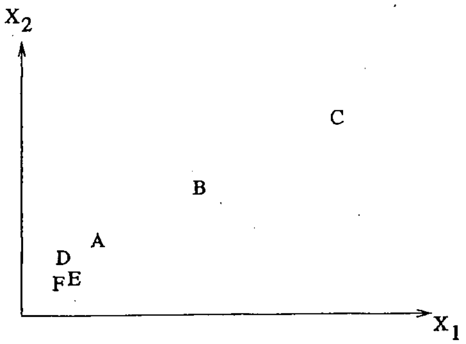


Figure 5. After a change in context, B and C are more similar than B and A.

$$s(\mathbf{x}_i, \mathbf{x}_j) = f(\mathbf{x}_i, \mathbf{x}_j, \mathscr{C}),$$

where $\mathscr{C}$ is the context (the set of surrounding points). One metric defined using context is the *mutual neighbor distance* (MND), proposed in Gowda and Krishna [1977], which is given by

$$MND(\mathbf{x}_i, \mathbf{x}_j) = NN(\mathbf{x}_i, \mathbf{x}_j) + NN(\mathbf{x}_j, \mathbf{x}_i),$$

where $NN(\mathbf{x}_i, \mathbf{x}_j)$ is the neighbor number of $\mathbf{x}_j$ with respect to $\mathbf{x}_i$. Figures 4 and 5 give an example. In Figure 4, the nearest neighbor of A is B, and B's nearest neighbor is A. So, $NN(A, B) = NN(B, A) = 1$ and the MND between A and B is 2. However, $NN(B, C) = 1$ but $NN(C, B) = 2$, and therefore $MND(B, C) = 3$. Figure 5 was obtained from Figure 4 by adding three new points D, E, and F. Now $MND(B, C) = 3$ (as before), but $MND(A, B) = 5$. The MND between A and B has increased by introducing additional points, even though A and B have not moved. The MND is not a metric (it does not satisfy the triangle inequality [Zhang 1995]). In spite of this, MND has been successfully applied in several clustering applications [Gowda and Diday 1992]. This observation supports the viewpoint that the dissimilarity does not need to be a metric.

Watanabe's theorem of the ugly duckling [Watanabe 1985] states:

> "*Insofar as we use a finite set of predicates that are capable of distinguishing any two objects considered, the number of predicates shared by any two such objects is constant, independent of the choice of objects.*"

This implies that it is possible to make any two arbitrary patterns equally similar by encoding them with a sufficiently large number of features. As a consequence, any two arbitrary patterns are equally similar, unless we use some additional domain information. For example, in the case of conceptual clustering [Michalski and Stepp 1983], the similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$ is defined as

$$s(\mathbf{x}_i, \mathbf{x}_j) = f(\mathbf{x}_i, \mathbf{x}_j, \mathscr{C}, \mathscr{E}),$$

where $\mathscr{C}$ is a set of pre-defined concepts. This notion is illustrated with the help of Figure 6. Here, the Euclidean distance between points A and B is less than that between B and C. However, B and C can be viewed as "more similar" than A and B because B and C belong to the same concept (ellipse) and A belongs to a different concept (rectangle). The conceptual similarity measure is the most general similarity measure. We

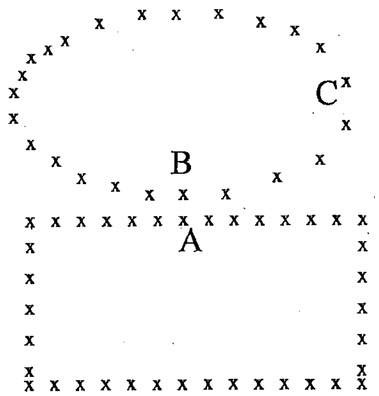


Figure 6. Conceptual similarity between points .

discuss several pragmatic issues associated with its use in Section 5.

5. CLUSTERING TECHNIQUES

Different approaches to clustering data can be described with the help of the hierarchy shown in Figure 7 (other taxonometric representations of clustering methodology are possible; ours is based on the discussion in Jain and Dubes [1988]). At the top level, there is a distinction between hierarchical and partitional approaches (hierarchical methods produce a nested series of partitions, while partitional methods produce only one).

The taxonomy shown in Figure 7 must be supplemented by a discussion of cross-cutting issues that may (in principle) affect all of the different approaches regardless of their placement in the taxonomy.

—Agglomerative *vs.* divisive: This aspect relates to algorithmic structure and operation. An agglomerative approach begins with each pattern in a distinct (singleton) cluster, and successively merges clusters together until a stopping criterion is satisfied. A divisive method begins with all patterns in a single cluster and performs splitting until a stopping criterion is met.

—Monothetic *vs.* polythetic: This aspect relates to the sequential or simultaneous use of features in the clustering process. Most algorithms are polythetic; that is, all features enter into the computation of distances between patterns, and decisions are based on those distances. A simple monothetic algorithm reported in Anderberg [1973] considers features sequentially to divide the given collection of patterns. This is illustrated in Figure 8. Here, the collection is divided into two groups using feature x_1; the vertical broken line V is the separating line. Each of these clusters is further divided independently using feature x_2, as depicted by the broken lines H_1 and H_2. The major problem with this algorithm is that it generates 2^d clusters where d is the dimensionality of the patterns. For large values of d ($d > 100$ is typical in information retrieval applications [Salton 1991]), the number of clusters generated by this algorithm is so large that the data set is divided into uninterestingly small and fragmented clusters.

—Hard *vs.* fuzzy: A hard clustering algorithm allocates each pattern to a single cluster during its operation and in its output. A fuzzy clustering method assigns degrees of membership in several clusters to each input pattern. A fuzzy clustering can be converted to a hard clustering by assigning each pattern to the cluster with the largest measure of membership.

—Deterministic *vs.* stochastic: This issue is most relevant to partitional approaches designed to optimize a squared error function. This optimization can be accomplished using traditional techniques or through a random search of the state space consisting of all possible labelings.

—Incremental *vs.* non-incremental: This issue arises when the pattern set

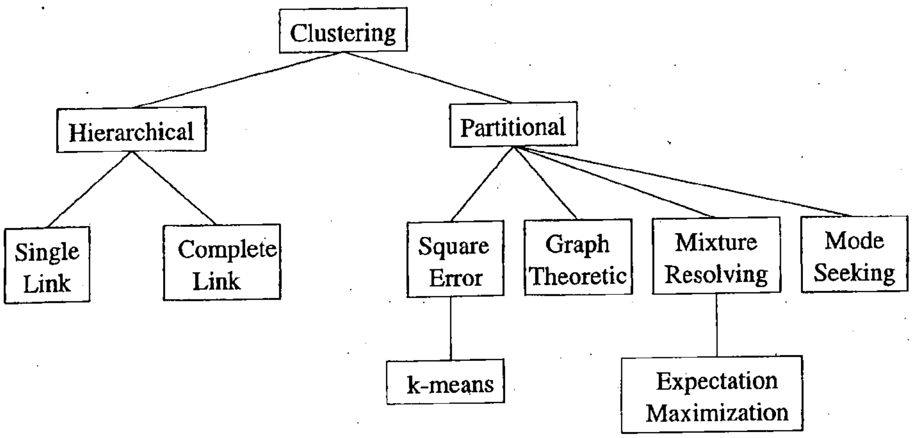


Figure 7. A taxonomy of clustering approaches.

to be clustered is large, and constraints on execution time or memory space affect the architecture of the algorithm. The early history of clustering methodology does not contain many examples of clustering algorithms designed to work with large data sets, but the advent of data mining has fostered the development of clustering algorithms that minimize the number of scans through the pattern set, reduce the number of patterns examined during execution, or reduce the size of data structures used in the algorithm's operations.

A cogent observation in Jain and Dubes [1988] is that the specification of an algorithm for clustering usually leaves considerable flexibilty in implementation.

5.1 Hierarchical Clustering Algorithms

The operation of a hierarchical clustering algorithm is illustrated using the two-dimensional data set in Figure 9. This figure depicts seven patterns labeled A, B, C, D, E, F, and G in three clusters. A hierarchical algorithm yields a *dendrogram* representing the nested grouping of patterns and similarity levels at which groupings change. A dendrogram corresponding to the seven points in Figure 9 (obtained from the single-link algorithm [Jain and Dubes 1988]) is shown in Figure 10. The dendrogram can be broken at different levels to yield different clusterings of the data.

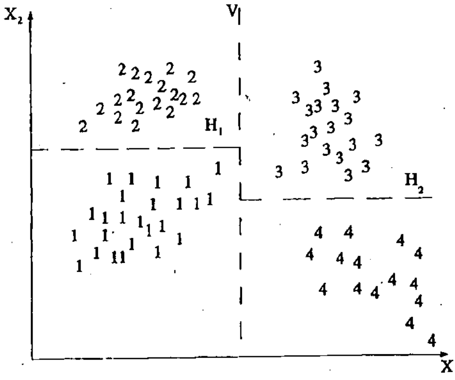


Figure 8. Monothetic partitional clustering.

Most hierarchical clustering algorithms are variants of the single-link [Sneath and Sokal 1973], complete-link [King 1967], and minimum-variance [Ward 1963; Murtagh 1984] algorithms. Of these, the single-link and complete-link algorithms are most popular. These two algorithms differ in the way they characterize the similarity between a pair of clusters. In the single-link method, the distance between two clus-

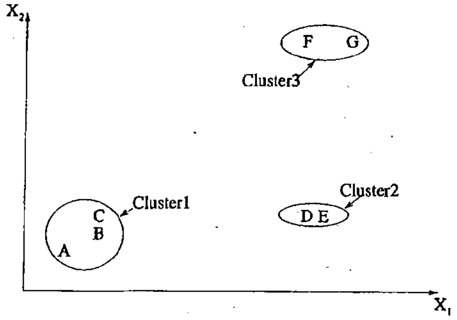


Figure 9. Points falling in three clusters.

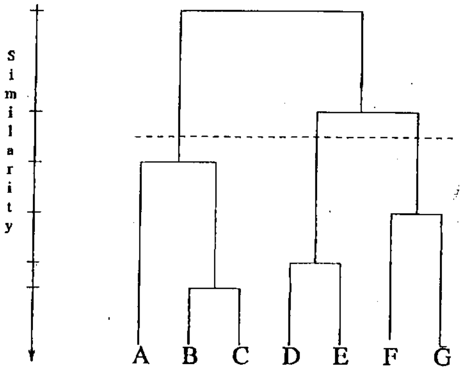


Figure 10. The dendrogram obtained using the single-link algorithm.

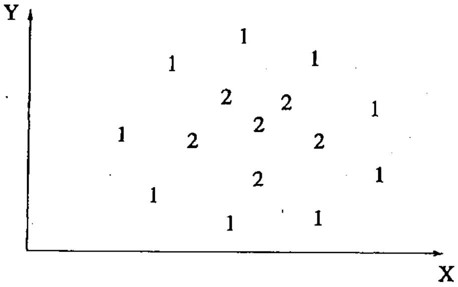


Figure 11. Two concentric clusters.

ters is the *minimum* of the distances between all pairs of patterns drawn from the two clusters (one pattern from the first cluster, the other from the second). In the complete-link algorithm, the distance between two clusters is the *maximum* of all pairwise distances between patterns in the two clusters. In either case, two clusters are merged to form a larger cluster based on minimum distance criteria. The complete-link algorithm produces tightly bound or compact clusters [Baeza-Yates 1992]. The single-link algorithm, by contrast, suffers from a chaining effect [Nagy 1968]. It has a tendency to produce clusters that are straggly or elongated. There are two clusters in Figures 12 and 13 separated by a "bridge" of noisy patterns. The single-link algorithm produces the clusters shown in Figure 12, whereas the complete-link algorithm obtains the clustering shown in Figure 13. The clusters obtained by the complete-link algorithm are more compact than those obtained by the single-link algorithm; the cluster labeled 1 obtained using the single-link algorithm is elongated because of the noisy patterns labeled "*". The single-link algorithm is more versatile than the complete-link algorithm, otherwise. For example, the single-link algorithm can extract the concentric clusters shown in Figure 11, but the complete-link algorithm cannot. However, from a pragmatic viewpoint, it has been observed that the complete-link algorithm produces more useful hierarchies in many applications than the single-link algorithm [Jain and Dubes 1988].

Agglomerative Single-Link Clustering Algorithm

(1) Place each pattern in its own cluster. Construct a list of interpattern distances for all distinct unordered pairs of patterns, and sort this list in ascending order.

(2) Step through the sorted list of distances, forming for each distinct dissimilarity value d_k a graph on the patterns where pairs of patterns closer than d_k are connected by a graph edge. If all the patterns are members of a connected graph, stop. Otherwise, repeat this step.

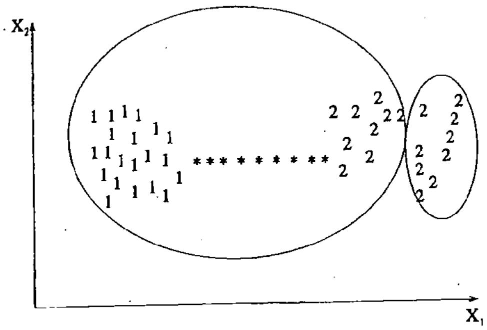


Figure 12. A single-link clustering of a pattern set containing two classes (1 and 2) connected by a chain of noisy patterns (*).

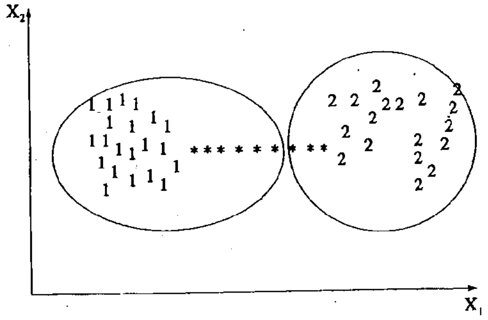


Figure 13. A complete-link clustering of a pattern set containing two classes (1 and 2) connected by a chain of noisy patterns (*).

(3) The output of the algorithm is a nested hierarchy of graphs which can be cut at a desired dissimilarity level forming a partition (clustering) identified by simply connected components in the corresponding graph.

Agglomerative Complete-Link Clustering Algorithm

(1) Place each pattern in its own cluster. Construct a list of interpattern distances for all distinct unordered pairs of patterns, and sort this list in ascending order.

(2) Step through the sorted list of distances, forming for each distinct dissimilarity value d_k a graph on the patterns where pairs of patterns closer than d_k are connected by a graph edge. If all the patterns are members of a completely connected graph, stop.

(3) The output of the algorithm is a nested hierarchy of graphs which can be cut at a desired dissimilarity level forming a partition (clustering) identified by completely connected components in the corresponding graph.

Hierarchical algorithms are more versatile than partitional algorithms. For example, the single-link clustering algorithm works well on data sets containing non-isotropic clusters including well-separated, chain-like, and concentric clusters, whereas a typical partitional algorithm such as the k-means algorithm works well only on data sets having isotropic clusters [Nagy 1968]. On the other hand, the time and space complexities [Day 1992] of the partitional algorithms are typically lower than those of the hierarchical algorithms. It is possible to develop hybrid algorithms [Murty and Krishna 1980] that exploit the good features of both categories.

Hierarchical Agglomerative Clustering Algorithm

(1) Compute the proximity matrix containing the distance between each pair of patterns. Treat each pattern as a cluster.

(2) Find the most similar pair of clusters using the proximity matrix. Merge these two clusters into one cluster. Update the proximity matrix to reflect this merge operation.

(3) If all patterns are in one cluster, stop. Otherwise, go to step 2.

Based on the way the proximity matrix is updated in step 2, a variety of agglomerative algorithms can be designed. Hierarchical divisive algorithms start with a single cluster of all the given objects and keep splitting the clusters based on some criterion to obtain a partition of singleton clusters.

5.2 Partitional Algorithms

A partitional clustering algorithm obtains a single partition of the data instead of a clustering structure, such as the dendrogram produced by a hierarchical technique. Partitional methods have advantages in applications involving large data sets for which the construction of a dendrogram is computationally prohibitive. A problem accompanying the use of a partitional algorithm is the choice of the number of desired output clusters. A seminal paper [Dubes 1987] provides guidance on this key design decision. The partitional techniques usually produce clusters by optimizing a criterion function defined either locally (on a subset of the patterns) or globally (defined over all of the patterns). Combinatorial search of the set of possible labelings for an optimum value of a criterion is clearly computationally prohibitive. In practice, therefore, the algorithm is typically run multiple times with different starting states, and the best configuration obtained from all of the runs is used as the output clustering.

5.2.1 *Squared Error Algorithms.* The most intuitive and frequently used criterion function in partitional clustering techniques is the squared error criterion, which tends to work well with isolated and compact clusters. The squared error for a clustering $\mathcal{L}$ of a pattern set $\mathcal{X}$ (containing K clusters) is

$$e^2(\mathcal{X}, \mathcal{L}) = \sum_{j=1}^{K} \sum_{i=1}^{n_j} \|\mathbf{x}_i^{(j)} - \mathbf{c}_j\|^2,$$

where $\mathbf{x}_i^{(j)}$ is the i^{th} pattern belonging to the j^{th} cluster and $\mathbf{c}_j$ is the centroid of the j^{th} cluster.

The k-means is the simplest and most commonly used algorithm employing a squared error criterion [McQueen 1967]. It starts with a random initial partition and keeps reassigning the patterns to clusters based on the similarity between the pattern and the cluster centers until a convergence criterion is met (e.g., there is no reassignment of any pattern from one cluster to another, or the squared error ceases to decrease significantly after some number of iterations). The k-means algorithm is popular because it is easy to implement, and its time complexity is $O(n)$, where n is the number of patterns. A major problem with this algorithm is that it is sensitive to the selection of the initial partition and may converge to a local minimum of the criterion function value if the initial partition is not properly chosen. Figure 14 shows seven two-dimensional patterns. If we start with patterns A, B, and C as the initial means around which the three clusters are built, then we end up with the partition {{A}, {B, C}, {D, E, F, G}} shown by ellipses. The squared error criterion value is much larger for this partition than for the best partition {{A, B, C}, {D, E}, {F, G}} shown by rectangles, which yields the global minimum value of the squared error criterion function for a clustering containing three clusters. The correct three-cluster solution is obtained by choosing, for example, A, D, and F as the initial cluster means.

Squared Error Clustering Method

(1) Select an initial partition of the patterns with a fixed number of clusters and cluster centers.

(2) Assign each pattern to its closest cluster center and compute the new cluster centers as the centroids of the clusters. Repeat this step until convergence is achieved, i.e., until the cluster membership is stable.

(3) Merge and split clusters based on some heuristic information, optionally repeating step 2.

k-Means Clustering Algorithm

(1) Choose k cluster centers to coincide with k randomly-chosen patterns or k randomly defined points inside the hypervolume containing the pattern set.

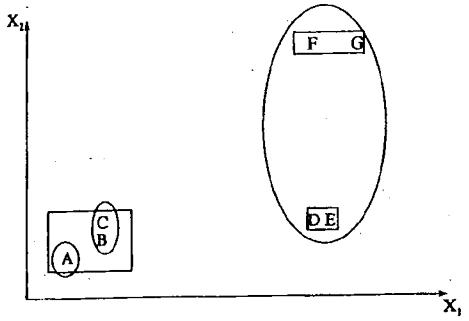


Figure 14. The k-means algorithm is sensitive to the initial partition.

(2) Assign each pattern to the closest cluster center.

(3) Recompute the cluster centers using the current cluster memberships.

(4) If a convergence criterion is not met, go to step 2. Typical convergence criteria are: no (or minimal) reassignment of patterns to new cluster centers, or minimal decrease in squared error.

Several variants [Anderberg 1973] of the k-means algorithm have been reported in the literature. Some of them attempt to select a good initial partition so that the algorithm is more likely to find the global minimum value.

Another variation is to permit splitting and merging of the resulting clusters. Typically, a cluster is split when its variance is above a pre-specified threshold, and two clusters are merged when the distance between their centroids is below another pre-specified threshold. Using this variant, it is possible to obtain the optimal partition starting from any arbitrary initial partition, provided proper threshold values are specified. The well-known ISODATA [Ball and Hall 1965] algorithm employs this technique of merging and splitting clusters. If ISODATA is given the "ellipse" partitioning shown in Figure 14 as an initial partitioning, it will produce the optimal three-cluster partitioning. ISODATA will first merge the clusters {A} and {B,C} into one cluster because the distance between their centroids is small and then split the cluster {D,E,F,G}, which has a large variance, into two clusters {D,E} and {F,G}.

Another variation of the k-means algorithm involves selecting a different criterion function altogether. The *dynamic clustering* algorithm (which permits representations other than the centroid for each cluster) was proposed in Diday [1973], and Symon [1977] and describes a dynamic clustering approach obtained by formulating the clustering problem in the framework of maximum-likelihood estimation. The regularized Mahalanobis distance was used in Mao and Jain [1996] to obtain hyperellipsoidal clusters.

5.2.2 *Graph-Theoretic Clustering.* The best-known graph-theoretic divisive clustering algorithm is based on construction of the *minimal spanning tree* (MST) of the data [Zahn 1971], and then deleting the MST edges with the largest lengths to generate clusters. Figure 15 depicts the MST obtained from nine two-dimensional points. By breaking the link labeled CD with a length of 6 units (the edge with the maximum Euclidean length), two clusters ({A, B, C} and {D, E, F, G, H, I}) are obtained. The second cluster can be further divided into two clusters by breaking the edge EF, which has a length of 4.5 units.

The hierarchical approaches are also related to graph-theoretic clustering. Single-link clusters are subgraphs of the minimum spanning tree of the data [Gower and Ross 1969] which are also the connected components [Gotlieb and Kumar 1968]. Complete-link clusters are maximal complete subgraphs, and are related to the node colorability of graphs [Backer and Hubert 1976]. The maximal complete subgraph was considered the strictest definition of a cluster in Augustson and Minker [1970] and Raghavan and Yu [1981]. A graph-oriented approach for non-hierarchical structures and overlapping clusters is

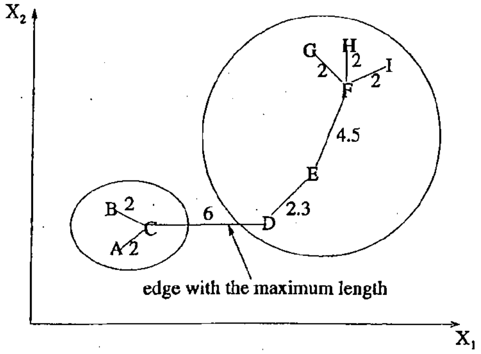


Figure 15. Using the minimal spanning tree to form clusters.

presented in Ozawa [1985]. The *Delaunay graph* (DG) is obtained by connecting all the pairs of points that are Voronoi neighbors. The DG contains all the neighborhood information contained in the MST and the relative neighborhood graph (RNG) [Toussaint 1980].

5.3 Mixture-Resolving and Mode-Seeking Algorithms

The mixture resolving approach to cluster analysis has been addressed in a number of ways. The underlying assumption is that the patterns to be clustered are drawn from one of several distributions, and the goal is to identify the parameters of each and (perhaps) their number. Most of the work in this area has assumed that the individual components of the mixture density are Gaussian, and in this case the parameters of the individual Gaussians are to be estimated by the procedure. Traditional approaches to this problem involve obtaining (iteratively) a maximum likelihood estimate of the parameter vectors of the component densities [Jain and Dubes 1988].

More recently, the Expectation Maximization (EM) algorithm (a general-purpose maximum likelihood algorithm [Dempster et al. 1977] for missing-data problems) has been applied to the problem of parameter estimation. A recent book [Mitchell 1997] provides an accessible description of the technique. In the EM framework, the parameters of the component densities are unknown, as are the mixing parameters, and these are estimated from the patterns. The EM procedure begins with an initial estimate of the parameter vector and iteratively rescores the patterns against the mixture density produced by the parameter vector. The rescored patterns are then used to update the parameter estimates. In a clustering context, the scores of the patterns (which essentially measure their likelihood of being drawn from particular components of the mixture) can be viewed as hints at the class of the pattern. Those patterns, placed (by their scores) in a particular component, would therefore be viewed as belonging to the same cluster.

Nonparametric techniques for density-based clustering have also been developed [Jain and Dubes 1988]. Inspired by the Parzen window approach to nonparametric density estimation, the corresponding clustering procedure searches for bins with large counts in a multidimensional histogram of the input pattern set. Other approaches include the application of another partitional or hierarchical clustering algorithm using a distance measure based on a nonparametric density estimate.

5.4 Nearest Neighbor Clustering

Since proximity plays a key role in our intuitive notion of a cluster, nearest-neighbor distances can serve as the basis of clustering procedures. An iterative procedure was proposed in Lu and Fu [1978]; it assigns each unlabeled pattern to the cluster of its nearest labeled neighbor pattern, provided the distance to that labeled neighbor is below a threshold. The process continues until all patterns are labeled or no additional labelings occur. The mutual neighborhood value (described earlier in the context of distance computation) can also be used to grow clusters from near neighbors.

5.5 Fuzzy Clustering

Traditional clustering approaches generate partitions; in a partition, each pattern belongs to one and only one cluster. Hence, the clusters in a hard clustering are disjoint. Fuzzy clustering extends this notion to associate each pattern with every cluster using a membership function [Zadeh 1965]. The output of such algorithms is a clustering, but not a partition. We give a high-level partitional fuzzy clustering algorithm below.

Fuzzy Clustering Algorithm

(1) Select an initial fuzzy partition of the N objects into K clusters by selecting the $N \times K$ membership matrix U. An element u_{ij} of this matrix represents the grade of membership of object $\mathbf{x}_i$ in cluster $\mathbf{c}_j$. Typically, $u_{ij} \in [0,1]$.

(2) Using U, find the value of a fuzzy criterion function, e.g., a weighted squared error criterion function, associated with the corresponding partition. One possible fuzzy criterion function is

$$E^2(\mathcal{X}, \mathbf{U}) = \sum_{i=1}^{N}\sum_{k=1}^{K} u_{ij}\|\mathbf{x}_i - \mathbf{c}_k\|^2,$$

where $\mathbf{c}_k = \sum_{i=1}^{N} u_{ik}\mathbf{x}_i$ is the k^{th} fuzzy cluster center.

Reassign patterns to clusters to reduce this criterion function value and recompute U.

(3) Repeat step 2 until entries in U do not change significantly.

In fuzzy clustering, each cluster is a fuzzy set of all the patterns. Figure 16 illustrates the idea. The rectangles enclose two "hard" clusters in the data: $H_1 = \{1,2,3,4,5\}$ and $H_2 = \{6,7,8,9\}$. A fuzzy clustering algorithm might produce the two fuzzy clusters F_1 and F_2 depicted by ellipses. The patterns will have membership values in [0,1] for each cluster. For example, fuzzy cluster F_1 could be compactly described as

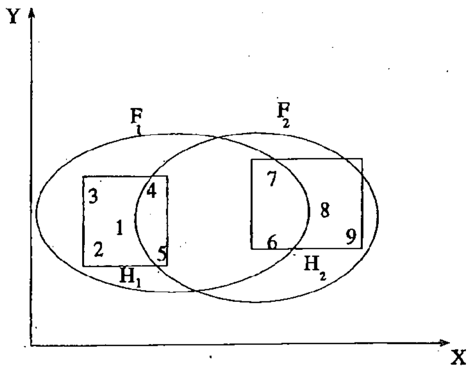


Figure 16. Fuzzy clusters.

$$\{(1,0.9), (2,0.8), (3,0.7), (4,0.6), (5,0.55), (6,0.2), (7,0.2), (8,0.0), (9,0.0)\}$$

and F_2 could be described as

$$\{(1,0.0), (2,0.0), (3,0.0), (4,0.1), (5,0.15), (6,0.4), (7,0.35), (8,1.0), (9,0.9)\}$$

The ordered pairs (i, μ_i) in each cluster represent the ith pattern and its membership value to the cluster μ_i. Larger membership values indicate higher confidence in the assignment of the pattern to the cluster. A hard clustering can be obtained from a fuzzy partition by thresholding the membership value.

Fuzzy set theory was initially applied to clustering in Ruspini [1969]. The book by Bezdek [1981] is a good source for material on fuzzy clustering. The most popular fuzzy clustering algorithm is the fuzzy c-means (FCM) algorithm. Even though it is better than the hard k-means algorithm at avoiding local minima, FCM can still converge to local minima of the squared error criterion. The design of membership functions is the most important problem in fuzzy clustering; different choices include

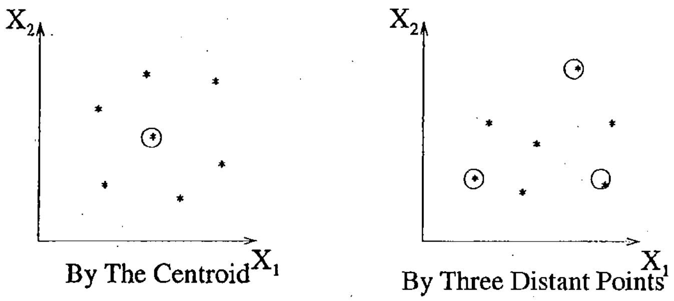


Figure 17. Representation of a cluster by points.

those based on similarity decomposition and centroids of clusters. A generalization of the FCM algorithm was proposed by Bezdek [1981] through a family of objective functions. A fuzzy c-shell algorithm and an adaptive variant for detecting circular and elliptical boundaries was presented in Dave [1992].

5.6 Representation of Clusters

In applications where the number of classes or clusters in a data set must be discovered, a partition of the data set is the end product. Here, a partition gives an idea about the separability of the data points into clusters and whether it is meaningful to employ a supervised classifier that assumes a given number of classes in the data set. However, in many other applications that involve decision making, the resulting clusters have to be represented or described in a compact form to achieve *data abstraction*. Even though the construction of a cluster representation is an important step in decision making, it has not been examined closely by researchers. The notion of cluster representation was introduced in Duran and Odell [1974] and was subsequently studied in Diday and Simon [1976] and Michalski et al. [1981]. They suggested the following representation schemes:

(1) Represent a cluster of points by their centroid or by a set of distant points in the cluster. Figure 17 depicts these two ideas.

(2) Represent clusters using nodes in a classification tree. This is illustrated in Figure 18.

(3) Represent clusters by using conjunctive logical expressions. For example, the expression $[X_1 > 3][X_2 < 2]$ in Figure 18 stands for the logical statement 'X_1 is greater than 3' *and* 'X_2 is less than 2'.

Use of the centroid to represent a cluster is the most popular scheme. It works well when the clusters are compact or isotropic. However, when the clusters are elongated or non-isotropic, then this scheme fails to represent them properly. In such a case, the use of a collection of boundary points in a cluster captures its shape well. The number of points used to represent a cluster should increase as the complexity of its shape increases. The two different representations illustrated in Figure 18 are equivalent. Every path in a classification tree from the root node to a leaf node corresponds to a conjunctive statement. An important limitation of the typical use of the simple conjunctive concept representations is that they can describe only rectangular or isotropic clusters in the feature space.

Data abstraction is useful in decision making because of the following:

(1) It gives a simple and intuitive description of clusters which is easy for human comprehension. In both conceptual clustering [Michalski

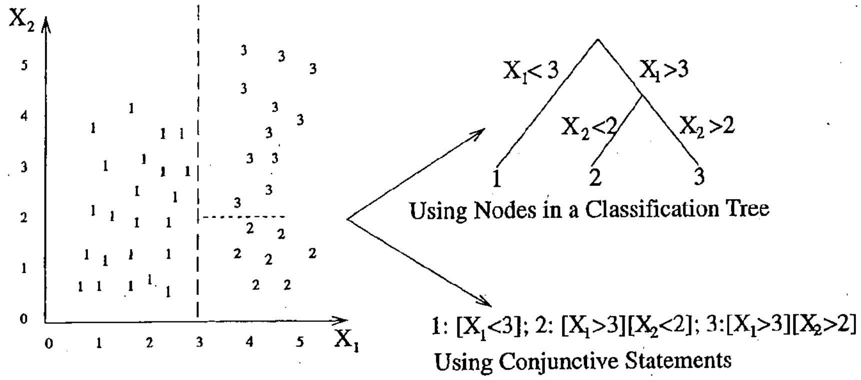

X₂
X₁
0 1 2 3 4 5
X₁< 3
X₁>3
X₂<2
X₂ >2
1 2 3
Using Nodes in a Classification Tree
1: [X₁<3]; 2: [X₁>3][X₂<2]; 3:[X₁>3][X₂>2]
Using Conjunctive Statements

Figure 18. Representation of clusters by a classification tree or by conjunctive statements.

and Stepp 1983] and symbolic clustering [Gowda and Diday 1992] this representation is obtained without using an additional step. These algorithms generate the clusters as well as their descriptions. A set of fuzzy rules can be obtained from fuzzy clusters of a data set. These rules can be used to build fuzzy classifiers and fuzzy controllers.

(2) It helps in achieving data compression that can be exploited further by a computer [Murty and Krishna 1980]. Figure 19(a) shows samples belonging to two chain-like clusters labeled 1 and 2. A partitional clustering like the k-means algorithm cannot separate these two structures properly. The single-link algorithm works well on this data, but is computationally expensive. So a hybrid approach may be used to exploit the desirable properties of both these algorithms. We obtain 8 subclusters of the data using the (computationally efficient) k-means algorithm. Each of these subclusters can be represented by their centroids as shown in Figure 19(a). Now the single-link algorithm can be applied on these centroids alone to cluster them into 2 groups. The resulting groups are shown in Figure 19(b). Here, a data reduction is achieved by representing the subclusters by their centroids.

(3) It increases the efficiency of the decision making task. In a cluster-based document retrieval technique [Salton 1991], a large collection of documents is clustered and each of the clusters is represented using its centroid. In order to retrieve documents relevant to a query, the query is matched with the cluster centroids rather than with all the documents. This helps in retrieving relevant documents efficiently. Also in several applications involving large data sets, clustering is used to perform indexing, which helps in efficient decision making [Dorai and Jain 1995].

5.7 Artificial Neural Networks for Clustering

Artificial neural networks (ANNs) [Hertz et al. 1991] are motivated by biological neural networks. ANNs have been used extensively over the past three decades for both classification and clustering [Sethi and Jain 1991; Jain and Mao 1994]. Some of the features of the ANNs that are important in pattern clustering are:

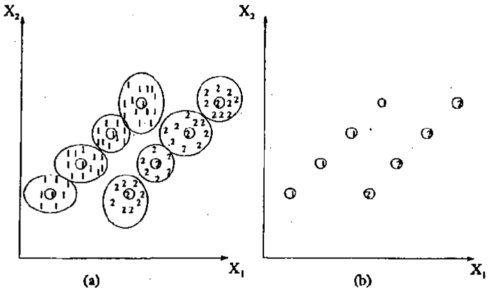


Figure 19. Data compression by clustering.

(1) ANNs process numerical vectors and so require patterns to be represented using quantitative features only.

(2) ANNs are inherently parallel and distributed processing architectures.

(3) ANNs may learn their interconnection weights adaptively [Jain and Mao 1996; Oja 1982]. More specifically, they can act as pattern normalizers and feature selectors by appropriate selection of weights.

Competitive (or winner–take–all) neural networks [Jain and Mao 1996] are often used to cluster input data. In competitive learning, similar patterns are grouped by the network and represented by a single unit (neuron). This grouping is done automatically based on data correlations. Well-known examples of ANNs used for clustering include Kohonen's learning vector quantization (LVQ) and self-organizing map (SOM) [Kohonen 1984], and adaptive resonance theory models [Carpenter and Grossberg 1990]. The architectures of these ANNs are simple: they are single-layered. Patterns are presented at the input and are associated with the output nodes. The weights between the input nodes and the output nodes are iteratively changed (this is called learning) until a termination criterion is satisfied. Competitive learning has been found to exist in biological neural networks. However, the learning or weight update procedures are quite similar to those in some classical clustering approaches. For example, the relationship between the k-means algorithm and LVQ is addressed in Pal et al. [1993]. The learning algorithm in ART models is similar to the leader clustering algorithm [Moor 1988].

The SOM gives an intuitively appealing two-dimensional map of the multidimensional data set, and it has been successfully used for vector quantization and speech recognition [Kohonen 1984]. However, like its sequential counterpart, the SOM generates a suboptimal partition if the initial weights are not chosen properly. Further, its convergence is controlled by various parameters such as the learning rate and a neighborhood of the winning node in which learning takes place. It is possible that a particular input pattern can fire different output units at different iterations; this brings up the *stability* issue of learning systems. The system is said to be stable if no pattern in the training data changes its category after a finite number of learning iterations. This problem is closely associated with the problem of *plasticity*, which is the ability of the algorithm to adapt to new data. For stability, the learning rate should be decreased to zero as iterations progress and this affects the plasticity. The ART models are supposed to be stable and plastic [Carpenter and Grossberg 1990]. However, ART nets are order-dependent; that is, different partitions are obtained for different orders in which the data is presented to the net. Also, the size and number of clusters generated by an ART net depend on the value chosen for the *vigilance threshold*, which is used to decide whether a pattern is to be assigned to one of the existing clusters or start a new cluster. Further, both SOM and ART are suitable for detecting only hyperspherical clusters [Hertz et al. 1991]. A two-layer network that employs regularized Mahalanobis distance to extract hyperellipsoidal clusters was proposed in Mao and Jain [1994]. All these ANNs use a fixed number of output nodes which limit the number of clusters that can be produced.

5.8 Evolutionary Approaches for Clustering

Evolutionary approaches, motivated by natural evolution, make use of evolutionary operators and a population of solutions to obtain the globally optimal partition of the data. Candidate solutions to the clustering problem are encoded as chromosomes. The most commonly used evolutionary operators are: selection, recombination, and mutation. Each transforms one or more input chromosomes into one or more output chromosomes. A fitness function evaluated on a chromosome determines a chromosome's likelihood of surviving into the next generation. We give below a high-level description of an evolutionary algorithm applied to clustering.

An Evolutionary Algorithm for Clustering

(1) Choose a random population of solutions. Each solution here corresponds to a valid k-partition of the data. Associate a fitness value with each solution. Typically, fitness is inversely proportional to the squared error value. A solution with a small squared error will have a larger fitness value.

(2) Use the evolutionary operators selection, recombination and mutation to generate the next population of solutions. Evaluate the fitness values of these solutions.

(3) Repeat step 2 until some termination condition is satisfied.

The best-known evolutionary techniques are genetic algorithms (GAs) [Holland 1975; Goldberg 1989], evolution strategies (ESs) [Schwefel 1981], and evolutionary programming (EP) [Fogel et al. 1965]. Out of these three approaches, GAs have been most frequently used in clustering. Typically, solutions are binary strings in GAs. In GAs, a selection operator propagates solutions from the current generation to the next generation based on their fitness. Selection employs a probabilistic scheme so that solutions with higher fitness have a higher probability of getting reproduced.

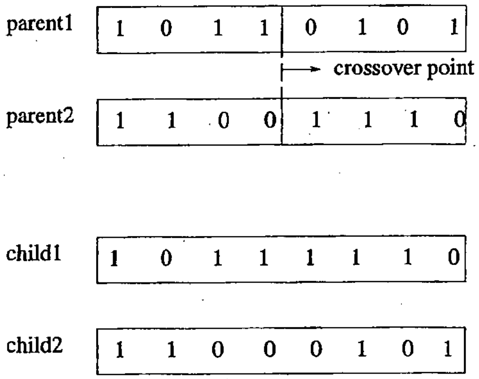


Figure 20. Crossover operation.

There are a variety of recombination operators in use; *crossover* is the most popular. Crossover takes as input a pair of chromosomes (called parents) and outputs a new pair of chromosomes (called children or offspring) as depicted in Figure 20. In Figure 20, a single point crossover operation is depicted. It exchanges the segments of the parents across a crossover point. For example, in Figure 20, the parents are the binary strings '10110101' and '11001110'. The segments in the two parents after the crossover point (between the fourth and fifth locations) are exchanged to produce the child chromosomes. *Mutation* takes as input a chromosome and outputs a chromosome by complementing the bit value at a randomly selected location in the input chromosome. For example, the string '11111110' is generated by applying the mutation operator to the second bit location in the string '10111110' (starting at the left). Both crossover and mutation are applied with some prespecified probabilities which depend on the fitness values.

GAs represent points in the search space as binary strings, and rely on the crossover operator to explore the search space. Mutation is used in GAs for the sake of completeness, that is, to make sure that no part of the search space is left unexplored. ESs and EP differ from the GAs in solution representation and type of the mutation operator used; EP does not use a recombination operator, but only selection and mutation. Each of these three approaches have been used to solve the clustering problem by viewing it as a minimization of the squared error criterion. Some of the theoretical issues such as the convergence of these approaches were studied in Fogel and Fogel [1994].

GAs perform a globalized search for solutions whereas most other clustering procedures perform a localized search. In a localized search, the solution obtained at the 'next iteration' of the procedure is in the vicinity of the current solution. In this sense, the k-means algorithm, fuzzy clustering algorithms, ANNs used for clustering, various annealing schemes (see below), and tabu search are all localized search techniques. In the case of GAs, the crossover and mutation operators can produce new solutions that are completely different from the current ones. We illustrate this fact in Figure 21. Let us assume that the scalar X is coded using a 5-bit binary representation, and let S_1 and S_2 be two points in the one-dimensional search space. The decimal values of S_1 and S_2 are 8 and 31, respectively. Their binary representations are S_1 = 01000 and S_2 = 11111. Let us apply the single-point crossover to these strings, with the crossover site falling between the second and third most significant bits as shown below.

01!000

11!111

This will produce a new pair of points or chromosomes S_3 and S_4 as shown in Figure 21. Here, S_3 = 01111 and S_4 = 11000. The corresponding decimal values are 15 and 24, respectively. Similarly, by mutating the most significant bit in the binary string 01111 (decimal 15), the binary string 11111 (decimal 31) is generated. These jumps, or gaps between points in successive generations, are much larger than those produced by other approaches.

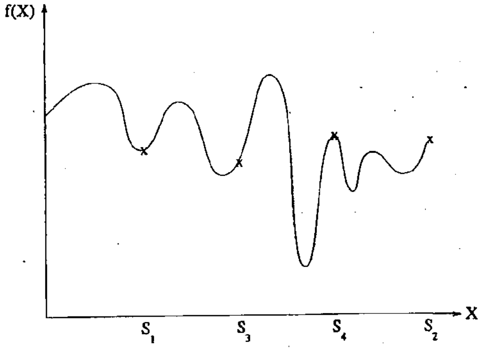


Figure 21. GAs perform globalized search.

Perhaps the earliest paper on the use of GAs for clustering is by Raghavan and Birchand [1979], where a GA was used to minimize the squared error of a clustering. Here, each point or chromosome represents a partition of N objects into K clusters and is represented by a K-ary string of length N. For example, consider six patterns—A, B, C, D, E, and F—and the string 101001. This six-bit binary (K = 2) string corresponds to placing the six patterns into two clusters. This string represents a two-partition, where one cluster has the first, third, and sixth patterns and the second cluster has the remaining patterns. In other words, the two clusters are {A,C,F} and {B,D,E} (the six-bit binary string 010110 represents the same clustering of the six patterns). When there are K clusters, there are K! different chromosomes corresponding to each K-partition of the data. This increases the effective search space size by a factor of K!. Further, if crossover is applied on two good chromosomes, the resulting offspring may be inferior in this representation. For example, let {A,B,C} and {D,E,F} be the clusters in the optimal 2-partition of the six patterns considered above. The corresponding chromosomes are 111000 and 000111. By applying single-point crossover at the location between the third and fourth bit positions on these two strings, we get 111111 and 000000 as offspring and both correspond to an inferior partition. These problems have motivated researchers to design better representation schemes and crossover operators.

In Bhuyan et al. [1991], an improved representation scheme is proposed where an additional separator symbol is used along with the pattern labels to represent a partition. Let the separator symbol be represented by *. Then the chromosome ACF*BDE corresponds to a 2-partition {A,C,F} and {B,D,E}. Using this representation permits them to map the clustering problem into a permutation problem such as the traveling salesman problem, which can be solved by using the permutation crossover operators [Goldberg 1989]. This solution also suffers from permutation redundancy. There are 72 equivalent chromosomes (permutations) corresponding to the same partition of the data into the two clusters {A,C,F} and {B,D,E}.

More recently, Jones and Beltramo [1991] investigated the use of edge-based crossover [Whitley et al. 1989] to solve the clustering problem. Here, all patterns in a cluster are assumed to form a complete graph by connecting them with edges. Offspring are generated from the parents so that they inherit the edges from their parents. It is observed that this crossover operator takes $O(K^6 + N)$ time for N patterns and K clusters ruling out its applicability on practical data sets having more than 10 clusters. In a hybrid approach proposed in Babu and Murty [1993], the GA is used only to find good initial cluster centers and the k-means algorithm is applied to find the final partition. This hybrid approach performed better than the GA.

A major problem with GAs is their sensitivity to the selection of various parameters such as population size, crossover and mutation probabilities, *etc.* Grefenstette [Grefenstette 1986] has studied this problem and suggested guidelines for selecting these control parameters. However, these guidelines may not yield good results on specific problems like pattern clustering. It was reported in Jones and Beltramo [1991] that hybrid genetic algorithms incorporating problem-specific heuristics are good for clustering. A similar claim is made in Davis [1991] about the applicability of GAs to other practical problems. Another issue with GAs is the selection of an appropriate representation which is low in order and short in defining length.

It is possible to view the clustering problem as an optimization problem that locates the optimal centroids of the clusters directly rather than finding an optimal partition using a GA. This view permits the use of ESs and EP, because centroids can be coded easily in both these approaches, as they support the direct representation of a solution as a real-valued vector. In Babu and Murty [1994], ESs were used on both hard and fuzzy clustering problems and EP has been used to evolve fuzzy min-max clusters [Fogel and Simpson 1993]. It has been observed that they perform better than their classical counterparts, the k-means algorithm and the fuzzy c-means algorithm. However, all of these approaches suffer (as do GAs and ANNs) from sensitivity to control parameter selection. For each specific problem, one has to tune the parameter values to suit the application.

5.9 Search-Based Approaches

Search techniques used to obtain the optimum value of the criterion function are divided into deterministic and stochastic search techniques. Deterministic search techniques guarantee an optimal partition by performing exhaustive enumeration. On the other hand, the stochastic search techniques generate a near-optimal partition reasonably quickly, and guarantee convergence to optimal partition asymptotically. Among the techniques considered so far, evolutionary approaches are stochastic and the remainder are deterministic. Other deterministic approaches to clustering include the branch-and-bound technique adopted in Koontz et al. [1975] and Cheng [1995] for generating optimal partitions. This approach generates the optimal partition of the data at the cost of excessive computational requirements. In Rose et al. [1993], a deterministic annealing approach was proposed for clustering. This approach employs an annealing technique in which the error surface is smoothed, but convergence to the global optimum is not guaranteed. The use of deterministic annealing in proximity-mode clustering (where the patterns are specified in terms of pairwise proximities rather than multidimensional points) was explored in Hofmann and Buhmann [1997]; later work applied the deterministic annealing approach to texture segmentation [Hofmann and Buhmann 1998].

The deterministic approaches are typically greedy descent approaches, whereas the stochastic approaches permit perturbations to the solutions in non-locally optimal directions also with nonzero probabilities. The stochastic search techniques are either sequential or parallel, while evolutionary approaches are inherently parallel. The simulated annealing approach (SA) [Kirkpatrick et al. 1983] is a sequential stochastic search technique, whose applicability to clustering is discussed in Klein and Dubes [1989]. Simulated annealing procedures are designed to avoid (or recover from) solutions which correspond to local optima of the objective functions. This is accomplished by accepting with some probability a new solution for the next iteration of lower quality (as measured by the criterion function). The probability of acceptance is governed by a critical parameter called the temperature (by analogy with annealing in metals), which is typically specified in terms of a starting (first iteration) and final temperature value. Selim and Al-Sultan [1991] studied the effects of control parameters on the performance of the algorithm, and Baeza-Yates [1992] used SA to obtain near-optimal partition of the data. SA is statistically guaranteed to find the global optimal solution [Aarts and Korst 1989]. A high-level outline of a SA based algorithm for clustering is given below.

Clustering Based on Simulated Annealing

(1) Randomly select an initial partition and P_0, and compute the squared error value, E_{P_0}. Select values for the control parameters, initial and final temperatures T_0 and T_f.

(2) Select a neighbor P_1 of P_0 and compute its squared error value, E_{P_1}. If E_{P_1} is larger than E_{P_0}, then assign P_1 to P_0 with a temperature-dependent probability. Else assign P_1 to P_0. Repeat this step for a fixed number of iterations.

(3) Reduce the value of T_0, *i.e.* $T_0 = cT_0$, where c is a predetermined constant. If T_0 is greater than T_f, then go to step 2. Else stop.

The SA algorithm can be slow in reaching the optimal solution, because optimal results require the temperature to be decreased very slowly from iteration to iteration.

Tabu search [Glover 1986], like SA, is a method designed to cross boundaries of feasibility or local optimality and to systematically impose and release constraints to permit exploration of otherwise forbidden regions. Tabu search was used to solve the clustering problem in Al-Sultan [1995].

5.10 A Comparison of Techniques

In this section we have examined various deterministic and stochastic search techniques to approach the clustering problem as an optimization problem. A majority of these methods use the squared error criterion function. Hence, the partitions generated by these approaches are not as versatile as those generated by hierarchical algorithms. The clusters generated are typically hyperspherical in shape. Evolutionary approaches are globalized search techniques, whereas the rest of the approaches are localized search technique. ANNs and GAs are inherently parallel, so they can be implemented using parallel hardware to improve their speed. Evolutionary approaches are population-based; that is, they search using more than one solution at a time, and the rest are based on using a single solution at a time. ANNs, GAs, SA, and Tabu search (TS) are all sensitive to the selection of various learning/control parameters. In theory, all four of these methods are weak methods [Rich 1983] in that they do not use explicit domain knowledge. An important feature of the evolutionary approaches is that they can find the optimal solution even when the criterion function is discontinuous.

An empirical study of the performance of the following heuristics for clustering was presented in Mishra and Raghavan [1994]; SA, GA, TS, randomized branch-and-bound (RBA) [Mishra and Raghavan 1994], and hybrid search (HS) strategies [Ismail and Kamel 1989] were evaluated. The conclusion was that GA performs well in the case of one-dimensional data, while its performance on high dimensional data sets is not impressive. The performance of SA is not attractive because it is very slow. RBA and TS performed best. HS is good for high dimensional data. However, none of the methods was found to be superior to others by a significant margin. An empirical study of k-means, SA, TS, and GA was presented in Al-Sultan and Khan [1996]. TS, GA and SA were judged comparable in terms of solution quality, and all were better than k-means. However, the k-means method is the most efficient in terms of execution time; other schemes took more time (by a factor of 500 to 2500) to partition a data set of size 60 into 5 clusters. Further, GA encountered the best solution faster than TS and SA; SA took more time than TS to encounter the best solution. However, GA took the maximum time for convergence, that is, to obtain a population of only the best solutions, followed by TS and SA. An important observation is that in both Mishra and Raghavan [1994] and Al-Sultan and Khan [1996] the sizes of the data sets considered are small; that is, fewer than 200 patterns.

A two-layer network was employed in Mao and Jain [1996], with the first layer including a number of principal component analysis subnets, and the second layer using a competitive net. This network performs partitional clustering using the regularized Mahalanobis distance. This net was trained using a set of 1000 randomly selected pixels from a large image and then used to classify every pixel in the image. Babu et al. [1997] proposed a stochastic connectionist approach (SCA) and compared its performance on standard data sets with both the SA and k-means algorithms. It was observed that SCA is superior to both SA and k-means in terms of solution quality. Evolutionary approaches are good only when the data size is less than 1000 and for low dimensional data..

In summary, only the k-means algorithm and its ANN equivalent, the Kohonen net [Mao and Jain 1996] have been applied on large data sets; other approaches have been tested, typically, on small data sets. This is because obtaining suitable learning/control parameters for ANNs, GAs, TS, and SA is difficult and their execution times are very high for large data sets. However, it has been shown [Selim and Ismail

1984] that the k-means method converges to a locally optimal solution. This behavior is linked with the initial seed selection in the k-means algorithm. So if a good initial partition can be obtained quickly using any of the other techniques, then k-means would work well even on problems with large data sets. Even though various methods discussed in this section are comparatively weak, it was revealed through experimental studies that combining domain knowledge would improve their performance. For example, ANNs work better in classifying images represented using extracted features than with raw images, and hybrid classifiers work better than ANNs [Mohiuddin and Mao 1994]. Similarly, using domain knowledge to hybridize a GA improves its performance [Jones and Beltramo 1991]. So it may be useful in general to use domain knowledge along with approaches like GA, SA, ANN, and TS. However, these approaches (specifically, the criteria functions used in them) have a tendency to generate a partition of hyperspherical clusters, and this could be a limitation. For example, in cluster-based document retrieval, it was observed that the hierarchical algorithms performed better than the partitional algorithms [Rasmussen 1992].

5.11 Incorporating Domain Constraints in Clustering

As a task, clustering is subjective in nature. The same data set may need to be partitioned differently for different purposes. For example, consider a *whale*, an *elephant*, and a *tuna fish* [Watanabe 1985]. Whales and elephants form a cluster of *mammals*. However, if the user is interested in partitioning them based on the concept of *living in water*, then whale and tuna fish are clustered together. Typically, this subjectivity is incorporated into the clustering criterion by incorporating domain knowledge in one or more phases of clustering.

Every clustering algorithm uses some type of knowledge either implicitly or explicitly. Implicit knowledge plays a role in (1) selecting a pattern representation scheme (e.g., using one's prior experience to select and encode features), (2) choosing a similarity measure (e.g., using the Mahalanobis distance instead of the Euclidean distance to obtain hyperellipsoidal clusters), and (3) selecting a grouping scheme (e.g., specifying the k-means algorithm when it is known that clusters are hyperspherical). Domain knowledge is used implicitly in ANNs, GAs, TS, and SA to select the control/learning parameter values that affect the performance of these algorithms.

It is also possible to use explicitly available domain knowledge to constrain or guide the clustering process. Such specialized clustering algorithms have been used in several applications. Domain concepts can play several roles in the clustering process, and a variety of choices are available to the practitioner. At one extreme, the available domain concepts might easily serve as an additional feature (or several), and the remainder of the procedure might be otherwise unaffected. At the other extreme, domain concepts might be used to confirm or veto a decision arrived at independently by a traditional clustering algorithm, or used to affect the computation of distance in a clustering algorithm employing proximity. The incorporation of domain knowledge into clustering consists mainly of ad hoc approaches with little in common; accordingly, our discussion of the idea will consist mainly of motivational material and a brief survey of past work. Machine learning research and pattern recognition research intersect in this topical area, and the interested reader is referred to the prominent journals in machine learning (e.g., *Machine Learning*, *J. of AI Research*, or *Artificial Intelligence*) for a fuller treatment of this topic.

As documented in Cheng and Fu [1985], rules in an expert system may be clustered to reduce the size of the knowledge base. This modification of clustering was also explored in the domains of universities, congressional voting records, and terrorist events by Lebowitz [1987].

5.11.1 *Similarity Computation.* Conceptual knowledge was used explicitly in the similarity computation phase in Michalski and Stepp [1983]. It was assumed that the pattern representations were available and the dynamic clustering algorithm [Diday 1973] was used to group patterns. The clusters formed were described using conjunctive statements in predicate logic. It was stated in Stepp and Michalski [1986] and Michalski and Stepp [1983] that the groupings obtained by the conceptual clustering are superior to those obtained by the numerical methods for clustering. A critical analysis of that work appears in Dale [1985], and it was observed that monothetic divisive clustering algorithms generate clusters that can be described by conjunctive statements. For example, consider Figure 8. Four clusters in this figure, obtained using a monothetic algorithm, can be described by using conjunctive concepts as shown below:

Cluster 1: $[X \leq a] \wedge [Y \leq b]$

Cluster 2: $[X \leq a] \wedge [Y > b]$

Cluster 3: $[X > a] \wedge [Y > c]$

Cluster 4: $[X > a] \wedge [Y \leq c]$

where $\wedge$ is the Boolean conjunction ('and') operator, and a, b, and c are constants.

5.11.2 *Pattern Representation.* It was shown in Srivastava and Murty [1990] that by using knowledge in the pattern representation phase, as is implicitly done in numerical taxonomy approaches, it is possible to obtain the same partitions as those generated by conceptual clustering. In this sense, conceptual clustering and numerical taxonomy are not diametrically opposite, but are equivalent. In the case of conceptual clustering, domain knowledge is explicitly used in interpattern similarity computation, whereas in numerical taxonomy it is implicitly assumed that pattern representations are obtained using the domain knowledge.

5.11.3 *Cluster Descriptions.* Typically, in knowledge-based clustering, both the clusters and their descriptions or characterizations are generated [Fisher and Langley 1985]. There are some exceptions, for instance,, Gowda and Diday [1992], where only clustering is performed and no descriptions are generated explicitly. In conceptual clustering, a cluster of objects is described by a conjunctive logical expression [Michalski and Stepp 1983]. Even though a conjunctive statement is one of the most common descriptive forms used by humans, it is a limited form. In Shekar et al. [1987], functional knowledge of objects was used to generate more intuitively appealing cluster descriptions that employ the Boolean *implication* operator. A system that represents clusters probabilistically was described in Fisher [1987]; these descriptions are more general than conjunctive concepts, and are well-suited to hierarchical classification domains (e.g., the animal species hierarchy). A conceptual clustering system in which clustering is done first is described in Fisher and Langley [1985]. These clusters are then described using probabilities. A similar scheme was described in Murty and Jain [1995], but the descriptions are logical expressions that employ both conjunction and disjunction.

An important characteristic of conceptual clustering is that it is possible to group objects represented by both qualitative and quantitative features if the clustering leads to a conjunctive concept. For example, the concept *cricket ball* might be represented as

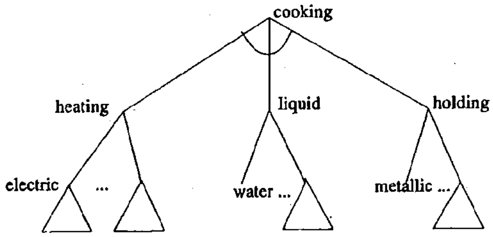


Figure 22. Functional knowledge.

$$\begin{aligned} \text{color} = \text{red} &\wedge (\text{shape} = \text{sphere}) \\ &\wedge (\text{make} = \text{leather}) \\ &\wedge (\text{radius} = 1.4 \text{ inches}), \end{aligned}$$

where radius is a quantitative feature and the rest are all qualitative features. This description is used to describe a cluster of cricket balls. In Stepp and Michalski [1986], a graph (the goal-dependency network) was used to group structured objects. In Shekar et al. [1987] functional knowledge was used to group man-made objects. Functional knowledge was represented using and/or trees [Rich 1983]. For example, the function *cooking* shown in Figure 22 can be decomposed into functions like *holding* and *heating* the material in a *liquid* medium. Each man-made object has a primary function for which it is produced. Further, based on its features, it may serve additional functions. For example, a book is meant for *reading*, but if it is heavy then it can also be used as a *paper weight*. In Sutton et al. [1993], object functions were used to construct generic recognition systems.

5.11.4 *Pragmatic Issues*. Any implementation of a system that explicitly incorporates domain concepts into a clustering technique has to address the following important pragmatic issues:

(1) Representation, availability and completeness of domain concepts.

(2) Construction of inferences using the knowledge.

(3) Accommodation of changing or dynamic knowledge.

In some domains, complete knowledge is available explicitly. For example, the *ACM Computing Reviews* classification tree used in Murty and Jain [1995] is complete and is explicitly available for use. In several domains, knowledge is incomplete and is not available explicitly. Typically, machine learning techniques are used to automatically extract knowledge, which is a difficult and challenging problem. The most prominently used learning method is "learning from examples" [Quinlan 1990]. This is an inductive learning scheme used to acquire knowledge from examples of each of the classes in different domains. Even if the knowledge is available explicitly, it is difficult to find out whether it is complete and sound. Further, it is extremely difficult to verify soundness and completeness of knowledge extracted from practical data sets, because such knowledge cannot be represented in propositional logic. It is possible that both the data and knowledge keep changing with time. For example, in a library, new books might get added and some old books might be deleted from the collection with time. Also, the classification system (knowledge) employed by the library is updated periodically.

A major problem with knowledge-based clustering is that it has not been applied to large data sets or in domains with large knowledge bases. Typically, the number of objects grouped was less than 1000, and number of rules used as a part of the knowledge was less than 100. The most difficult problem is to use a very large knowledge base for clustering objects in several practical problems including data mining, image segmentation, and document retrieval.

5.12 Clustering Large Data Sets

There are several applications where it is necessary to cluster a large collection of patterns. The definition of 'large' has varied (and will continue to do so) with changes in technology (e.g., memory and processing time). In the 1960s, 'large' meant several thousand patterns [Ross 1968]; now, there are applications where millions of patterns of high dimensionality have to be clustered. For example, to segment an image of size 500 × 500 pixels, the number of pixels to be clustered is 250,000. In document retrieval and information filtering, millions of patterns with a dimensionality of more than 100 have to be clustered to achieve data abstraction. A majority of the approaches and algorithms proposed in the literature cannot handle such large data sets. Approaches based on genetic algorithms, tabu search and simulated annealing are optimization techniques and are restricted to reasonably small data sets. Implementations of conceptual clustering optimize some criterion functions and are typically computationally expensive.

The convergent k-means algorithm and its ANN equivalent, the Kohonen net, have been used to cluster large data sets [Mao and Jain 1996]. The reasons behind the popularity of the k-means algorithm are:

(1) Its time complexity is $O(nkl)$, where n is the number of patterns, k is the number of clusters, and l is the number of iterations taken by the algorithm to converge. Typically, k and l are fixed in advance and so the algorithm has linear time complexity in the size of the data set [Day 1992].

(2) Its space complexity is $O(k + n)$. It requires additional space to store the data matrix. It is possible to store the data matrix in a secondary memory and access each pattern based on need. However, this scheme requires a huge access time because of the iterative nature of the algorithm, and as a consequence processing time increases enormously.

(3) It is order-independent; for a given initial seed set of cluster centers, it generates the same partition of the data irrespective of the order in which the patterns are presented to the algorithm.

Table I. Complexity of Clustering Algorithms

Clustering Algorithm	Time Complexity	Space Complexity
leader	$O(kn)$	$O(k)$
k-means	$O(nkl)$	$O(k)$
ISODATA	$O(nkl)$	$O(k)$
shortest spanning path	$O(n^2)$	$O(n)$
single-line	$O(n^2 \log n)$	$O(n^2)$
complete-line	$O(n^2 \log n)$	$O(n^2)$

However, the k-means algorithm is sensitive to initial seed selection and even in the best case, it can produce only hyperspherical clusters.

Hierarchical algorithms are more versatile. But they have the following disadvantages:

(1) The time complexity of hierarchical agglomerative algorithms is $O(n^2 \log n)$ [Kurita 1991]. It is possible to obtain single-link clusters using an MST of the data, which can be constructed in $O(n \log^2 n)$ time for two-dimensional data [Choudhury and Murty 1990].

(2) The space complexity of agglomerative algorithms is $O(n^2)$. This is because a similarity matrix of size $n \times n$ has to be stored. To cluster every pixel in a 100 × 100 image, approximately 200 megabytes of storage would be required (assuning single-precision storage of similarities). It is possible to compute the entries of this matrix based on need instead of storing them (this would increase the algorithm's time complexity [Anderberg 1973]).

Table I lists the time and space complexities of several well-known algorithms. Here, n is the number of patterns to be clustered, k is the number of clusters, and l is the number of iterations.

A possible solution to the problem of clustering large data sets while only marginally sacrificing the versatility of clusters is to implement more efficient variants of clustering algorithms. A hybrid approach was used in Ross [1968], where a set of reference points is chosen as in the k-means algorithm, and each of the remaining data points is assigned to one or more reference points or clusters. Minimal spanning trees (MST) are obtained for each group of points separately. These MSTs are merged to form an approximate global MST. This approach computes similarities between only a fraction of all possible pairs of points. It was shown that the number of similarities computed for 10,000 patterns using this approach is the same as the total number of pairs of points in a collection of 2,000 points. Bentley and Friedman [1978] contains an algorithm that can compute an approximate MST in $O(n \log n)$ time. A scheme to generate an approximate dendrogram incrementally in $O(n \log n)$ time was presented in Zupan [1982], while Venkateswarlu and Raju [1992] proposed an algorithm to speed up the ISODATA clustering algorithm. A study of the approximate single-linkage cluster analysis of large data sets was reported in Eddy et al. [1994]. In that work, an approximate MST was used to form single-link clusters of a data set of size 40,000.

The emerging discipline of data mining (discussed as an application in Section 6) has spurred the development of new algorithms for clustering large data sets. Two algorithms of note are the CLARANS algorithm developed by Ng and Han [1994] and the BIRCH algorithm proposed by Zhang et al. [1996]. CLARANS (Clustering Large Applications based on RANdom Search) identifies candidate cluster centroids through analysis of repeated random samples from the original data. Because of the use of random sampling, the time complexity is $O(n)$ for a pattern set of n elements. The BIRCH algorithm (Balanced Iterative Reducing and Clustering) stores summary information about candidate clusters in a dynamic tree data structure. This tree hierarchically organizes the clusterings represented at the leaf nodes. The tree can be rebuilt when a threshold specifying cluster size is updated manually, or when memory constraints force a change in this threshold. This algorithm, like CLARANS, has a time complexity linear in the number of patterns.

The algorithms discussed above work on large data sets, where it is possible to accommodate the entire pattern set in the main memory. However, there are applications where the entire data set cannot be stored in the main memory because of its size. There are currently three possible approaches to solve this problem.

(1) The pattern set can be stored in a secondary memory and subsets of this data clustered independently, followed by a merging step to yield a clustering of the entire pattern set. We call this approach the *divide and conquer* approach.

(2) An incremental clustering algorithm can be employed. Here, the entire data matrix is stored in a secondary memory and data items are transferred to the main memory one at a time for clustering. Only the cluster representations are stored in the main memory to alleviate the space limitations.

(3) A parallel implementation of a clustering algorithm may be used. We discuss these approaches in the next three subsections.

5.12.1 *Divide and Conquer Approach.* Here, we store the entire pattern matrix of size $n \times d$ in a secondary storage space (e.g., a disk file). We divide this data into p blocks, where an optimum value of p can be chosen based on the clustering algorithm used [Murty and Krishna 1980]. Let us assume that we have n/p patterns in each of the blocks.

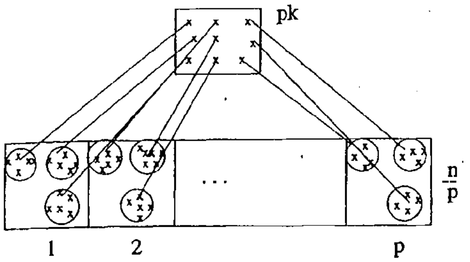


Figure 23. Divide and conquer approach to clustering.

We transfer each of these blocks to the main memory and cluster it into k clusters using a standard algorithm. One or more representative samples from each of these clusters are stored separately; we have pk of these representative patterns if we choose one representative per cluster. These pk representatives are further clustered into k clusters and the cluster labels of these representative patterns are used to relabel the original pattern matrix. We depict this two-level algorithm in Figure 23. It is possible to extend this algorithm to any number of levels; more levels are required if the data set is very large and the main memory size is very small [Murty and Krishna 1980]. If the single-link algorithm is used to obtain 5 clusters, then there is a substantial savings in the number of computations as shown in Table II for optimally chosen p when the number of clusters is fixed at 5. However, this algorithm works well only when the points in each block are reasonably homogeneous which is often satisfied by image data.

A two-level strategy for clustering a data set containing 2,000 patterns was described in Stahl [1986]. In the first level, the data set is loosely clustered into a large number of clusters using the leader algorithm. Representatives from these clusters, one per cluster, are the input to the second level clustering, which is obtained using Ward's hierarchical method.

Table II. Number of Distance Computations (n) for the Single-Link Clustering Algorithm and a Two-Level Divide and Conquer Algorithm

n	Single-link	p	Two-level
100	4,950		1200
500	124,750	2	10,750
100	499,500	4	31,500
10,000	49,995,000	10	1,013,750

5.12.2 *Incremental Clustering.* Incremental clustering is based on the assumption that it is possible to consider patterns one at a time and assign them to existing clusters. Here, a new data item is assigned to a cluster without affecting the existing clusters significantly. A high level description of a typical incremental clustering algorithm is given below.

An Incremental Clustering Algorithm

(1) Assign the first data item to a cluster.

(2) Consider the next data item. Either assign this item to one of the existing clusters or assign it to a new cluster. This assignment is done based on some criterion, *e.g.* the distance between the new item and the existing cluster centroids.

(3) Repeat step 2 till all the data items are clustered.

The major advantage with the incremental clustering algorithms is that it is not necessary to store the entire pattern matrix in the memory. So, the space requirements of incremental algorithms are very small. Typically, they are noniterative. So their time requirements are also small. There are several incremental clustering algorithms:

(1) The leader clustering algorithm [Hartigan 1975] is the simplest in terms of time complexity which is $O(nk)$. It has gained popularity because of its neural network implementation, the ART network [Carpenter and Grossberg 1990]. It is very easy to implement as it requires only $O(k)$ space.

(2) The shortest spanning path (SSP) algorithm [Slagle et al. 1975] was originally proposed for data reorganization and was successfully used in automatic auditing of records [Lee et al. 1978]. Here, SSP algorithm was used to cluster 2000 patterns using 18 features. These clusters are used to estimate missing feature values in data items and to identify erroneous feature values.

(3) The *cobweb* system [Fisher 1987] is an incremental conceptual clustering algorithm. It has been successfully used in engineering applications [Fisher et al. 1993].

(4) An incremental clustering algorithm for dynamic information processing was presented in Can [1993]. The motivation behind this work is that, in dynamic databases, items might get added and deleted over time. These changes should be reflected in the partition generated without significantly affecting the current clusters. This algorithm was used to cluster incrementally an INSPEC database of 12,684 documents corresponding to computer science and electrical engineering.

Order-independence is an important property of clustering algorithms. An algorithm is *order-independent* if it generates the same partition for any order in which the data is presented. Otherwise, it is *order-dependent*. Most of the incremental algorithms presented above are order-dependent. We illustrate this order-dependent property in Figure 24 where there are 6 two-dimensional objects labeled 1 to 6. If we present these patterns to the leader algorithm in the order 2,1,3,5,4,6 then the two clusters obtained are shown by ellipses. If the order is 1,2,6,4,5,3, then we get a two-partition as shown by the triangles. The SSP algorithm, *cobweb*, and the algorithm in Can [1993] are all order-dependent.

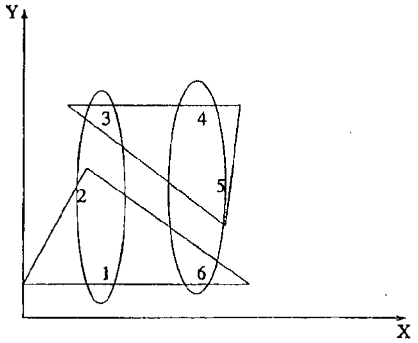


Figure 24. The leader algorithm is order dependent.

5.12.3 *Parallel Implementation.* Recent work [Judd et al. 1996] demonstrates that a combination of algorithmic enhancements to a clustering algorithm and distribution of the computations over a network of workstations can allow an entire 512 × 512 image to be clustered in a few minutes. Depending on the clustering algorithm in use, parallelization of the code and replication of data for efficiency may yield large benefits. However, a global shared data structure, namely the cluster membership table, remains and must be managed centrally or replicated and synchronized periodically. The presence or absence of robust, efficient parallel clustering techniques will determine the success or failure of cluster analysis in large-scale data mining applications in the future.

6. APPLICATIONS

Clustering algorithms have been used in a large variety of applications [Jain and Dubes 1988; Rasmussen 1992; Oehler and Gray 1995; Fisher et al. 1993]. In this section, we describe several applications where clustering has been employed as an essential step. These areas are: (1) image segmentation, (2) object and character recognition, (3) document retrieval, and (4) data mining.

6.1 Image Segmentation Using Clustering

Image segmentation is a fundamental component in many computer vision

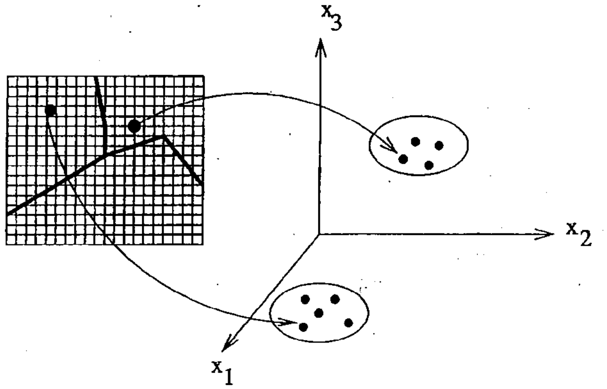


Figure 25. Feature representation for clustering. Image measurements and positions are transformed to features. Clusters in feature space correspond to image segments.

applications, and can be addressed as a clustering problem [Rosenfeld and Kak 1982]. The segmentation of the image(s) presented to an image analysis system is critically dependent on the scene to be sensed, the imaging geometry, configuration, and sensor used to transduce the scene into a digital image, and ultimately the desired output (goal) of the system.

The applicability of clustering methodology to the image segmentation problem was recognized over three decades ago, and the paradigms underlying the initial pioneering efforts are still in use today. A recurring theme is to define feature vectors at every image location (pixel) composed of both functions of image intensity and functions of the pixel location itself. This basic idea, depicted in Figure 25, has been successfully used for intensity images (with or without texture), range (depth) images and multispectral images.

6.1.1 *Segmentation.* An *image segmentation* is typically defined as an exhaustive partitioning of an input image into regions, each of which is considered to be homogeneous with respect to some image property of interest (e.g., intensity, color, or texture) [Jain et al. 1995]. If

$$\mathcal{I} = \{x_{ij}, i = 1 \ldots N_r, j = 1 \ldots N_c\}$$

is the input image with N_r rows and N_c columns and measurement value x_{ij} at pixel (i, j), then the segmentation can be expressed as $\mathcal{S} = \{S_1, \ldots S_k\}$, with the lth segment

$$S_l = \{(i_{l_1}, j_{l_1}), \ldots (i_{l_{N_l}}, j_{l_{N_l}})\}$$

consisting of a connected subset of the pixel coordinates. No two segments share any pixel locations ($S_i \cap S_j = \emptyset \; \forall i \neq j$), and the union of all segments covers the entire image ($\cup_{i=1}^{k} S_i = \{1 \ldots N_r\} \times \{1 \ldots N_c\}$). Jain and Dubes [1988], after Fu and Mui [1981] identified three techniques for producing segmentations from input imagery: *region-based*, *edge-based*, or *cluster-based*.

Consider the use of simple gray level thresholding to segment a high-contrast intensity image. Figure 26(a) shows a grayscale image of a textbook's bar code scanned on a flatbed scanner. Part b shows the results of a simple thresholding operation designed to separate the dark and light regions in the bar code area. Binarization steps like this are often performed in character recognition systems. Thresholding in effect 'clusters' the image pixels into two groups based on the one-dimensional intensity measurement [Rosenfeld 1969;

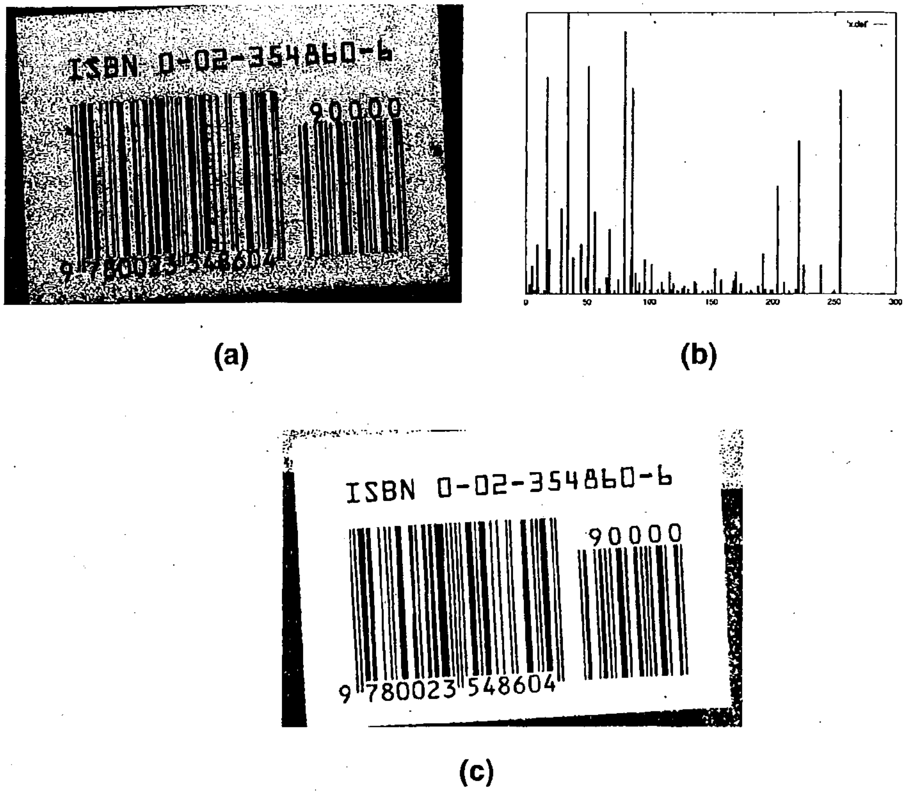


Figure 26. Binarization via thresholding. (a): Original grayscale image. (b): Gray-level histogram. (c): Results of thresholding.

Dunn et al. 1974]. A postprocessing step separates the classes into connected regions. While simple gray level thresholding is adequate in some carefully controlled image acquisition environments and much research has been devoted to appropriate methods for thresholding [Weszka 1978; Trier and Jain 1995], complex images require more elaborate segmentation techniques.

Many segmenters use measurements which are both *spectral* (e.g., the multispectral scanner used in remote sensing) and *spatial* (based on the pixel's location in the image plane). The measurement at each pixel hence corresponds directly to our concept of a pattern.

6.1.2 *Image Segmentation Via Clustering.* The application of local feature clustering to segment gray–scale images was documented in Schachter et al. [1979]. This paper emphasized the appropriate selection of features at each pixel rather than the clustering methodology, and proposed the use of image plane coordinates (spatial information) as additional features to be employed in clustering-based segmentation. The goal of clustering was to obtain a sequence of hyperellipsoidal clusters starting with cluster centers positioned at maximum density locations in the pattern space, and growing clusters about these centers until a χ^2 test for goodness of fit was violated. A variety of features were discussed and applied to both grayscale and color imagery.

An agglomerative clustering algorithm was applied in Silverman and Cooper [1988] to the problem of unsupervised learning of clusters of *coefficient vectors* for two image models that correspond to image segments. The first image model is polynomial for the observed image measurements; the assumption here is that the image is a collection of several adjoining graph surfaces, each a polynomial function of the image plane coordinates, which are sampled on the raster grid to produce the observed image. The algorithm proceeds by obtaining vectors of coefficients of least-squares fits to the data in M disjoint image windows. An agglomerative clustering algorithm merges (at each step) the two clusters that have a minimum global between-cluster Mahalanobis distance. The same framework was applied to segmentation of textured images, but for such images the polynomial model was inappropriate, and a parameterized Markov Random Field model was assumed instead.

Wu and Leahy [1993] describe the application of the principles of network flow to unsupervised classification, yielding a novel hierarchical algorithm for clustering. In essence, the technique views the unlabeled patterns as nodes in a graph, where the weight of an edge (i.e., its capacity) is a measure of similarity between the corresponding nodes. Clusters are identified by removing edges from the graph to produce connected disjoint subgraphs. In image segmentation, pixels which are 4-neighbors or 8-neighbors in the image plane share edges in the constructed adjacency graph, and the weight of a graph edge is based on the strength of a hypothesized image edge between the pixels involved (this strength is calculated using simple derivative masks). Hence, this segmenter works by finding closed contours in the image, and is best labeled edge-based rather than region-based.

In Vinod et al. [1994], two neural networks are designed to perform pattern clustering when combined. A two-layer network operates on a multidimensional histogram of the data to identify 'prototypes' which are used to classify the input patterns into clusters. These prototypes are fed to the classification network, another two-layer network operating on the histogram of the input data, but are trained to have differing weights from the prototype selection network. In both networks, the histogram of the image is used to weight the contributions of patterns neighboring the one under consideration to the location of prototypes or the ultimate classification; as such, it is likely to be more robust when compared to techniques which assume an underlying parametric density function for the pattern classes. This architecture was tested on gray-scale and color segmentation problems.

Jolion et al. [1991] describe a process for extracting clusters sequentially from the input pattern set by identifying hyperellipsoidal regions (bounded by loci of constant Mahalanobis distance) which contain a specified fraction of the unclassified points in the set. The extracted regions are compared against the best-fitting multivariate Gaussian density through a Kolmogorov-Smirnov test, and the fit quality is used as a figure of merit for selecting the 'best' region at each iteration. The process continues until a stopping criterion is satisfied. This procedure was applied to the problems of threshold selection for multithreshold segmentation of intensity imagery and segmentation of range imagery.

Clustering techniques have also been successfully used for the segmentation of range images, which are a popular source of input data for three-dimensional object recognition systems [Jain and Flynn 1993]. Range sensors typically return raster images with the measured value at each pixel being the coordinates of a 3D location in space. These 3D positions can be understood as the locations where rays emerging from the image plane locations in a bundle intersect the objects in front of the sensor.

The local feature clustering concept is particularly attractive for range image segmentation since (unlike intensity measurements) the measurements at each pixel have the same units (length); this would make ad hoc transformations or normalizations of the image features unnecessary if their goal is to impose equal scaling on those features. However, range image segmenters often add additional measurements to the feature space, removing this advantage.

A range image segmentation system described in Hoffman and Jain [1987] employs squared error clustering in a six-dimensional feature space as a source of an "initial" segmentation which is refined (typically by merging segments) into the output segmentation. The technique was enhanced in Flynn and Jain [1991] and used in a recent systematic comparison of range image segmenters [Hoover et al. 1996]; as such, it is probably one of the longest-lived range segmenters which has performed well on a large variety of range images.

This segmenter works as follows. At each pixel (i, j) in the input range image, the corresponding 3D measurement is denoted (x_{ij}, y_{ij}, z_{ij}), where typically x_{ij} is a linear function of j (the column number) and y_{ij} is a linear function of i (the row number). A $k \times k$ neighborhood of (i, j) is used to estimate the 3D surface normal $\mathbf{n}_{ij} = (n^x_{ij}, n^y_{ij}, n^z_{ij})$ at (i, j), typically by finding the least-squares planar fit to the 3D points in the neighborhood. The feature vector for the pixel at (i, j) is the six-dimensional measurement $(x_{ij}, y_{ij}, z_{ij}, n^x_{ij}, n^y_{ij}, n^z_{ij})$, and a candidate segmentation is found by clustering these feature vectors. For practical reasons, not every pixel's feature vector is used in the clustering procedure; typically 1000 feature vectors are chosen by subsampling.

The CLUSTER algorithm [Jain and Dubes 1988] was used to obtain segment labels for each pixel. CLUSTER is an enhancement of the k-means algorithm; it has the ability to identify several clusterings of a data set, each with a different number of clusters. Hoffman and Jain [1987] also experimented with other clustering techniques (e.g., complete-link, single-link, graph-theoretic, and other squared error algorithms) and found CLUSTER to provide the best combination of performance and accuracy. An additional advantage of CLUSTER is that it produces a sequence of output clusterings (i.e., a 2-cluster solution up through a K_{max}-cluster solution where K_{max} is specified by the user and is typically 20 or so); each clustering in this sequence yields a clustering statistic which combines between-cluster separation and within-cluster scatter. The clustering that optimizes this statistic is chosen as the best one. Each pixel in the range image is assigned the segment label of the nearest cluster center. This minimum distance classification step is not guaranteed to produce segments which are connected in the image plane; therefore, a connected components labeling algorithm allocates new labels for disjoint regions that were placed in the same cluster. Subsequent operations include surface type tests, merging of adjacent patches using a test for the presence of crease or jump edges between adjacent segments, and surface parameter estimation.

Figure 27 shows this processing applied to a range image. Part a of the figure shows the input range image; part b shows the distribution of surface normals. In part c, the initial segmentation returned by CLUSTER and modified to guarantee connected segments is shown. Part d shows the final segmentation produced by merging adjacent patches which do not have a significant crease edge between them. The final clusters reasonably represent distinct surfaces present in this complex object.

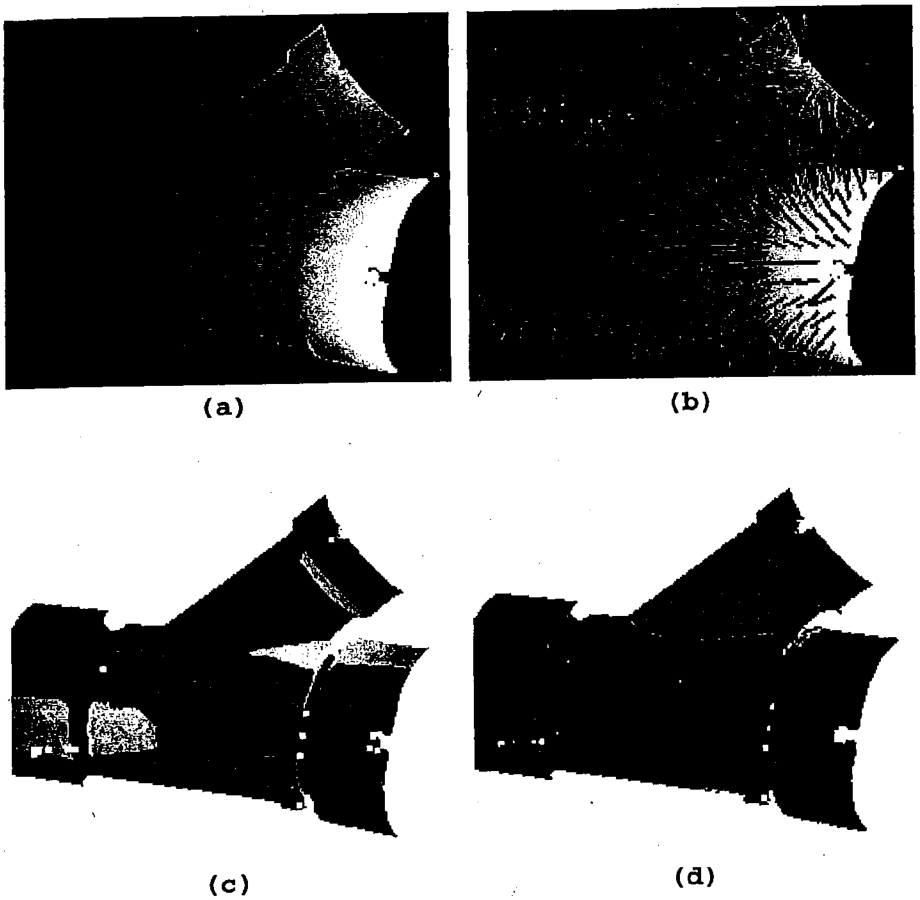


Figure 27. Range image segmentation using clustering. (a): Input range image. (b): Surface normals for selected image pixels. (c): Initial segmentation (19 cluster solution) returned by CLUSTER using 1000 six-dimensional samples from the image as a pattern set. (d): Final segmentation (8 segments) produced by postprocessing.

The analysis of textured images has been of interest to researchers for several years. Texture segmentation techniques have been developed using a variety of texture models and image operations. In Nguyen and Cohen [1993], texture image segmentation was addressed by modeling the image as a hierarchy of two Markov Random Fields, obtaining some simple statistics from each image block to form a feature vector, and clustering these blocks using a fuzzy K-means clustering method. The clustering procedure here is modified to jointly estimate the number of clusters as well as the fuzzy membership of each feature vector to the various clusters.

A system for segmenting texture images was described in Jain and Farrokhnia [1991]; there, Gabor filters were used to obtain a set of 28 orientation- and scale-selective features that characterize the texture in the neighborhood of each pixel. These 28 features are reduced to a smaller number through a feature selection procedure, and the resulting features are preprocessed and then clustered using the CLUSTER program. An index statistic

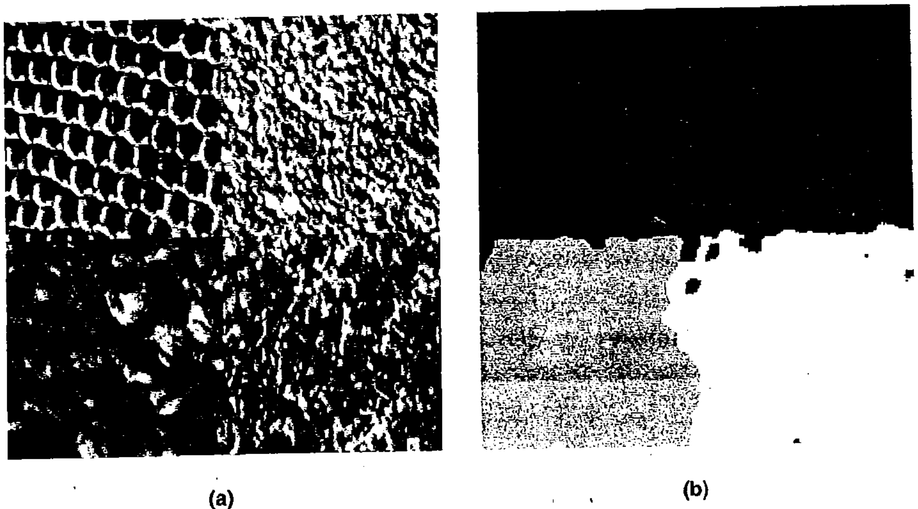


Figure 28. Texture image segmentation results. (a): Four-class texture mosaic. (b): Four-cluster solution produced by CLUSTER with pixel coordinates included in the feature set.

[Dubes 1987] is used to select the best clustering. Minimum distance classification is used to label each of the original image pixels. This technique was tested on several texture mosaics including the natural Brodatz textures and synthetic images. Figure 28(a) shows an input texture mosaic consisting of four of the popular Brodatz textures [Brodatz 1966]. Part b shows the segmentation produced when the Gabor filter features are augmented to contain spatial information (pixel coordinates). This Gabor filter based technique has proven very powerful and has been extended to the automatic segmentation of text in documents [Jain and Bhattacharjee 1992] and segmentation of objects in complex backgrounds [Jain et al. 1997].

Clustering can be used as a preprocessing stage to identify pattern classes for subsequent supervised classification. Taxt and Lundervold [1994] and Lundervold et al. [1996] describe a partitional clustering algorithm and a manual labeling technique to identify material classes (e.g., cerebrospinal fluid, white matter, striated muscle, tumor) in registered images of a human head obtained at five different magnetic resonance imaging channels (yielding a five-dimensional feature vector at each pixel). A number of clusterings were obtained and combined with domain knowledge (human expertise) to identify the different classes. Decision rules for supervised classification were based on these obtained classes. Figure 29(a) shows one channel of an input multispectral image; part b shows the 9-cluster result.

The k-means algorithm was applied to the segmentation of LANDSAT imagery in Solberg et al. [1996]. Initial cluster centers were chosen interactively by a trained operator, and correspond to land-use classes such as urban areas, soil (vegetation-free) areas, forest, grassland, and water. Figure 30(a) shows the input image rendered as grayscale; part b shows the result of the clustering procedure.

6.1.3 *Summary.* In this section, the application of clustering methodology to image segmentation problems has been motivated and surveyed. The historical record shows that clustering is a powerful tool for obtaining classifications of image pixels. Key issues in the design of any clustering-based segmenter are the

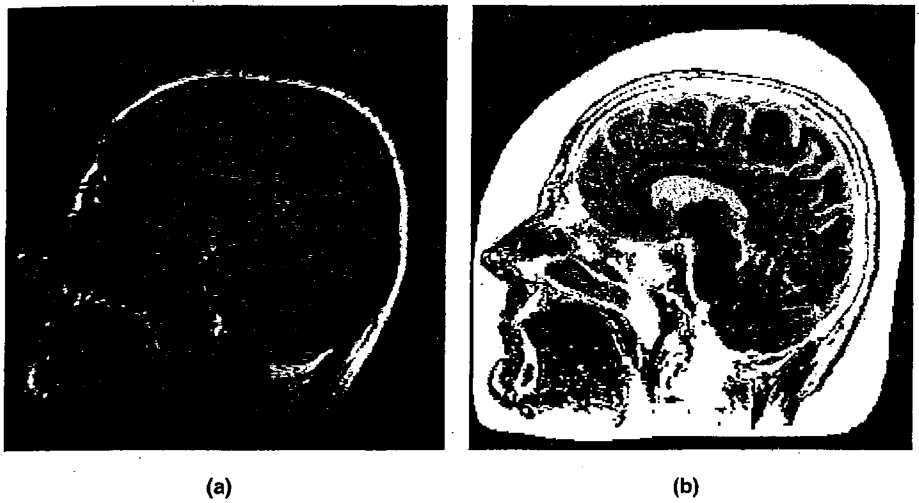


Figure 29. Multispectral medical image segmentation. (a): A single channel of the input image. (b): 9-cluster segmentation.

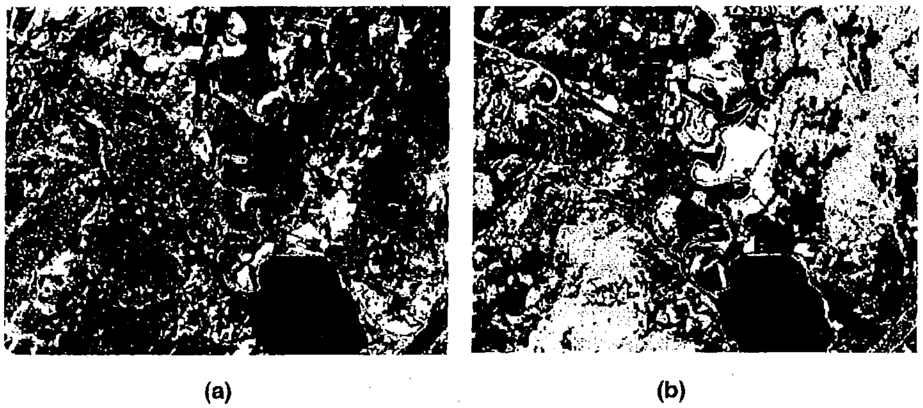


Figure 30. LANDSAT image segmentation. (a): Original image (ESA/EURIMAGE/Sattelitbild). (b): Clustered scene.

choice of pixel measurements (features) and dimensionality of the feature vector (i.e., should the feature vector contain intensities, pixel positions, model parameters, filter outputs?), a measure of similarity which is appropriate for the selected features and the application domain, the identification of a clustering algorithm, the development of strategies for feature and data reduction (to avoid the "curse of dimensionality" and the computational burden of classifying large numbers of patterns and/or features), and the identification of necessary pre- and post-processing techniques (e.g., image smoothing and minimum distance classification). The use of clustering for segmentation dates back to the 1960s, and new variations continue to emerge in the literature. Challenges to the more successful use of clustering include the high computational complexity of many clustering algorithms and their incorporation of strong assumptions (often multivariate Gaussian) about the multidimensional shape of clusters to be obtained. The ability of new clustering procedures to handle concepts and semantics in classification (in addition to numerical measurements) will be important for certain applications [Michalski and Stepp 1983; Murty and Jain 1995].

6.2 Object and Character Recognition

6.2.1 *Object Recognition.* The use of clustering to group views of 3D objects for the purposes of object recognition in range data was described in Dorai and Jain [1995]. The term *view* refers to a range image of an unoccluded object obtained from any arbitrary viewpoint. The system under consideration employed a *viewpoint dependent* (or view-centered) approach to the object recognition problem; each object to be recognized was represented in terms of a library of range images of that object.

There are many possible views of a 3D object and one goal of that work was to avoid matching an unknown input view against each image of each object. A common theme in the object recognition literature is *indexing*, wherein the unknown view is used to select a subset of views of a subset of the objects in the database for further comparison, and rejects all other views of objects. One of the approaches to indexing employs the notion of *view classes*; a view class is the set of qualitatively similar views of an object. In that work, the view classes were identified by clustering; the rest of this subsection outlines the technique.

Object views were grouped into classes based on the similarity of shape spectral features. Each input image of an object viewed in isolation yields a feature vector which characterizes that view. The feature vector contains the first ten central moments of a normalized shape spectral distribution, $\bar{H}(h)$, of an object view. The shape spectrum of an object view is obtained from its range data by constructing a histogram of *shape index* values (which are related to surface curvature values) and accumulating all the object pixels that fall into each bin. By normalizing the spectrum with respect to the total object area, the scale (size) differences that may exist between different objects are removed. The first moment m_1 is computed as the weighted mean of $\bar{H}(h)$:

$$m_1 = \sum_h (h)\bar{H}(h). \qquad (1)$$

The other central moments, m_p, $2 \le p \le 10$ are defined as:

$$m_p = \sum_h (h - m_1)^p \bar{H}(h). \qquad (2)$$

Then, the feature vector is denoted as $R = (m_1, m_2, \cdots, m_{10})$, with the range of each of these moments being $[-1,1]$.

Let $\mathcal{O} = \{O^1, O^2, \cdots, O^n\}$ be a collection of n 3D objects whose views are present in the model database, $\mathcal{M}_D$. The ith view of the jth object, O^i_j in the database is represented by $\langle L^i_j, R^i_j \rangle$, where L^i_j is the object label and R^i_j is the feature vector. Given a set of object representations $\mathfrak{R}^i = \{\langle L^i_1, R^i_1 \rangle, \cdots, \langle L^i_m, R^i_m \rangle\}$ that describes m views of the ith object, the goal is to derive a partition of the views, $\mathcal{P}^i = \{C^i_1, C^i_2, \cdots, C^i_{k_i}\}$. Each cluster in $\mathcal{P}^i$ contains those views of the ith object that have been adjudged similar based on the dissimilarity between the corresponding moment features of the shape spectra of the views. The measure of dissimilarity, between R^i_j and R^i_k, is defined as:

$$\mathfrak{D}(R^i_j, R^i_k) = \sum_{l=1}^{10} (R^i_{jl} - R^i_{kl})^2. \qquad (3)$$

6.2.2 *Clustering Views.* A database containing 3,200 range images of 10 different sculpted objects with 320 views per object is used [Dorai and Jain 1995].

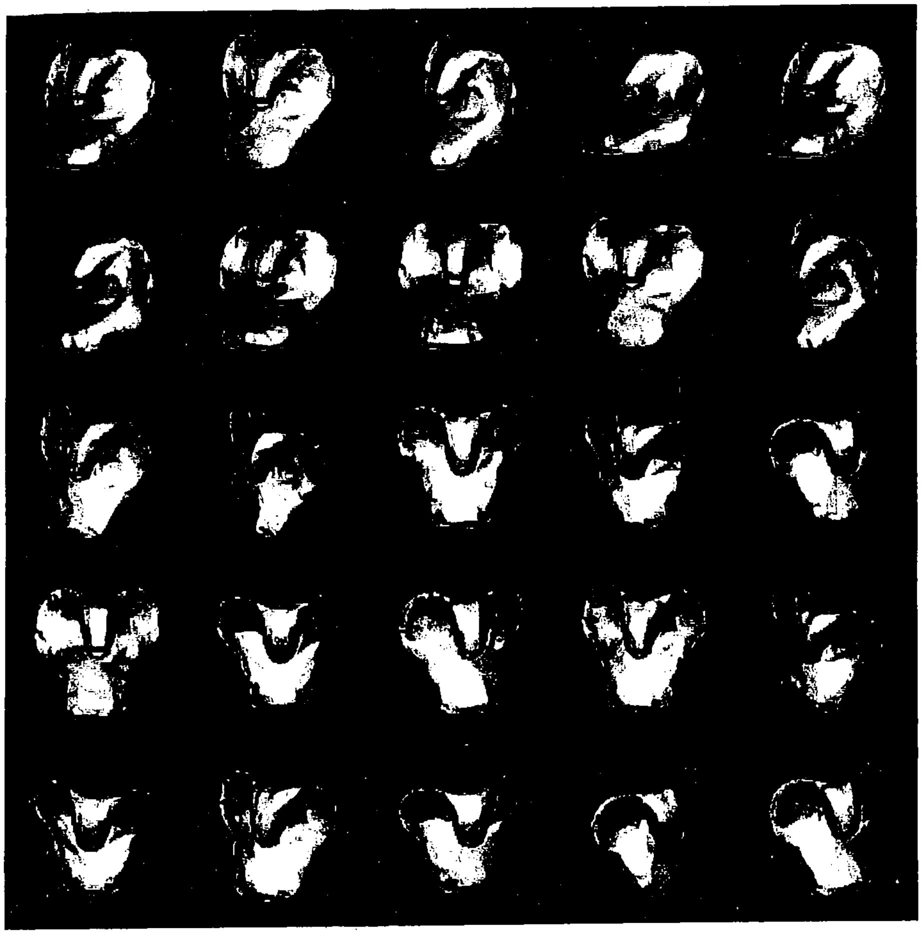


Figure 31. A subset of views of Cobra chosen from a set of 320 views.

The range images from 320 possible viewpoints (determined by the tessellation of the view-sphere using the icosahedron) of the objects were synthesized. Figure 31 shows a subset of the collection of views of Cobra used in the experiment.

The shape spectrum of each view is computed and then its feature vector is determined. The views of each object are clustered, based on the dissimilarity measure $\mathfrak{D}$ between their moment vectors using the complete-link hierarchical clustering scheme [Jain and Dubes 1988]. The hierarchical grouping obtained with 320 views of the Cobra object is shown in Figure 32. The view grouping hierarchies of the other nine objects are similar to the dendrogram in Figure 32. This dendrogram is cut at a dissimilarity level of 0.1 or less to obtain compact and well-separated clusters. The clusterings obtained in this manner demonstrate that the views of each object fall into several distinguishable clusters. The centroid of each of these clusters was determined by computing the mean of the moment vectors of the views falling into the cluster.

Dorai and Jain [1995] demonstrated that this clustering-based view grouping procedure facilitates object matching

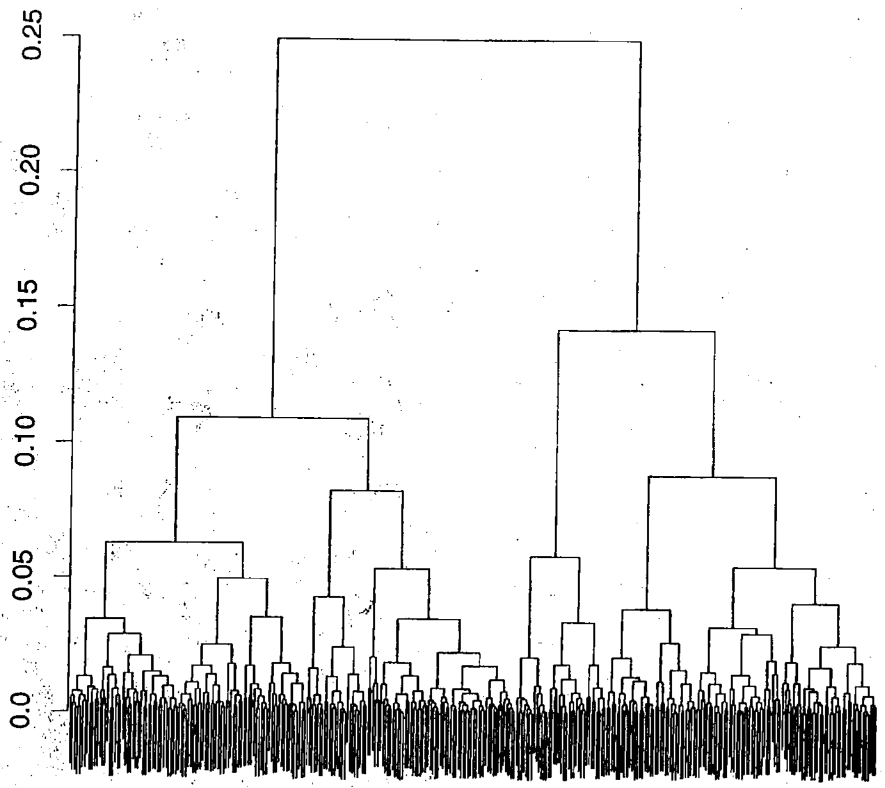


Figure 32. Hierarchical grouping of 320 views of a cobra sculpture.

in terms of classification accuracy and the number of matches necessary for correct classification of test views. Object views are grouped into compact and homogeneous view clusters, thus demonstrating the power of the cluster-based scheme for view organization and efficient object matching.

6.2.3 *Character Recognition.* Clustering was employed in Connell and Jain [1998] to identify lexemes in handwritten text for the purposes of writer-independent handwriting recognition. The success of a handwriting recognition system is vitally dependent on its acceptance by potential users. Writer-dependent systems provide a higher level of recognition accuracy than writer-independent systems, but require a large amount of training data. A writer-independent system, on the other hand, must be able to recognize a wide variety of writing styles in order to satisfy an individual user. As the variability of the writing styles that must be captured by a system increases, it becomes more and more difficult to discriminate between different classes due to the amount of overlap in the feature space. One solution to this problem is to separate the data from these disparate writing styles for each class into different subclasses, known as *lexemes*. These lexemes represent portions of the data which are more easily separated from the data of classes other than that to which the lexeme belongs.

In this system, handwriting is captured by digitizing the (x, y) position of the pen and the state of the pen point (up or down) at a constant sampling rate. Following some resampling, normalization, and smoothing, each stroke of the pen is represented as a variable-length string of points. A metric based on elastic template matching and dynamic programming is defined to allow the distance between two strokes to be calculated.

Using the distances calculated in this manner, a proximity matrix is constructed for each class of digits (i.e., 0 through 9). Each matrix measures the intraclass distances for a particular digit class. Digits in a particular class are clustered in an attempt to find a small number of prototypes. Clustering is done using the CLUSTER program described above [Jain and Dubes 1988], in which the feature vector for a digit is its N proximities to the digits of the same class. CLUSTER attempts to produce the best clustering for each value of K over some range, where K is the number of clusters into which the data is to be partitioned. As expected, the mean squared error (MSE) decreases monotonically as a function of K. The "optimal" value of K is chosen by identifying a "knee" in the plot of MSE *vs.* K.

When representing a cluster of digits by a single prototype, the best on-line recognition results were obtained by using the digit that is closest to that cluster's center. Using this scheme, a correct recognition rate of 99.33% was obtained.

6.3 Information Retrieval

Information retrieval (IR) is concerned with automatic storage and retrieval of documents [Rasmussen 1992]. Many university libraries use IR systems to provide access to books, journals, and other documents. Libraries use the Library of Congress Classification (LCC) scheme for efficient storage and retrieval of books. The LCC scheme consists of classes labeled A to Z [LC Classification Outline 1990] which are used to characterize books belonging to different subjects. For example, label Q corresponds to books in the area of science, and the subclass QA is assigned to mathematics. Labels QA76 to QA76.8 are used for classifying books related to computers and other areas of computer science.

There are several problems associated with the classification of books using the LCC scheme. Some of these are listed below:

(1) When a user is searching for books in a library which deal with a topic of interest to him, the LCC number alone may not be able to retrieve all the relevant books. This is because the classification number assigned to the books or the subject categories that are typically entered in the database do not contain sufficient information regarding all the topics covered in a book. To illustrate this point, let us consider the book *Algorithms for Clustering Data* by Jain and Dubes [1988]. Its LCC number is 'QA 278.J35'. In this LCC number, QA 278 corresponds to the topic 'cluster analysis', J corresponds to the first author's name and 35 is the serial number assigned by the Library of Congress. The subject categories for this book provided by the publisher (which are typically entered in a database to facilitate search) are cluster analysis, data processing and algorithms. There is a chapter in this book [Jain and Dubes 1988] that deals with computer vision, image processing, and image segmentation. So a user looking for literature on computer vision and, in particular, image segmentation will not be able to access this book by searching the database with the help of either the LCC number or the subject categories provided in the database. The LCC number for computer vision books is TA 1632 [LC Classification 1990] which is very different from the number QA 278.J35 assigned to this book.

(2) There is an inherent problem in assigning LCC numbers to books in a rapidly developing area. For example, let us consider the area of neural networks. Initially, category 'QP' in LCC scheme was used to label books and conference proceedings in this area. For example, Proceedings of the International Joint Conference on Neural Networks [IJCNN'91] was assigned the number 'QP 363.3'. But most of the recent books on neural networks are given a number using the category label 'QA'; Proceedings of the IJCNN'92 [IJCNN'92] is assigned the number 'QA 76.87'. Multiple labels for books dealing with the same topic will force them to be placed on different stacks in a library. Hence, there is a need to update the classification labels from time to time in an emerging discipline.

(3) Assigning a number to a new book is a difficult problem. A book may deal with topics corresponding to two or more LCC numbers, and therefore, assigning a unique number to such a book is difficult.

Murty and Jain [1995] describe a knowledge-based clustering scheme to group representations of books, which are obtained using the ACM CR (Association for Computing Machinery *Computing Reviews*) classification tree [ACM CR Classifications 1994]. This tree is used by the authors contributing to various ACM publications to provide keywords in the form of ACM CR category labels. This tree consists of 11 nodes at the first level. These nodes are labeled A to K. Each node in this tree has a label that is a string of one or more symbols. These symbols are alphanumeric characters. For example, I515 is the label of a fourth-level node in the tree.

6.3.1 *Pattern Representation.* Each book is represented as a generalized list [Sangal 1991] of these strings using the ACM CR classification tree. For the sake of brevity in representation, the fourth-level nodes in the ACM CR classification tree are labeled using numerals 1 to 9 and characters A to Z. For example, the children nodes of I.5.1 (*models*) are labeled I.5.1.1 to I.5.1.6. Here, I.5.1.1 corresponds to the node labeled *deterministic*, and I.5.1.6 stands for the node labeled *structural*. In a similar fashion, all the fourth-level nodes in the tree can be labeled as necessary. From now on, the dots in between successive symbols will be omitted to simplify the representation. For example, I.5.1.1 will be denoted as I511.

We illustrate this process of representation with the help of the book by Jain and Dubes [1988]. There are five chapters in this book. For simplicity of processing, we consider only the information in the chapter contents. There is a single entry in the table of contents for chapter 1, 'Introduction,' and so we do not extract any keywords from this. Chapter 2, labeled 'Data Representation,' has section titles that correspond to the labels of the nodes in the ACM CR classification tree [ACM CR Classifications, 1994] which are given below:

(1a) I522 (*feature evaluation and selection*),

(2b) I532 (*similarity measures*), and

(3c) I515 (*statistical*).

Based on the above analysis, Chapter 2 of Jain and Dubes [1988] can be characterized by the weighted disjunction ((I522 ∨ I532 ∨ I515)(1,4)). The weights (1,4) denote that it is one of the four chapters which plays a role in the representation of the book. Based on the table of contents, we can use one or more of the strings I522, I532, and I515 to represent Chapter 2. In a similar manner, we can represent other chapters in this book as weighted disjunctions based on the table of contents and the ACM CR classification tree. The representation of the entire book, the conjunction of all these chapter representations, is given by (((I522 ∨ I532 ∨ I515)(1,4) ∧ ((I515 ∨ I531)(2,4)) ∧ ((I541 ∨ I46 ∨ I434)(1,4))).

Currently, these representations are generated manually by scanning the table of contents of books in computer science area as ACM CR classification tree provides knowledge of computer science books only. The details of the collection of books used in this study are available in Murty and Jain [1995].

6.3.2 *Similarity Measure.* The similarity between two books is based on the similarity between the corresponding strings. Two of the well-known distance functions between a pair of strings are [Baeza-Yates 1992] the Hamming distance and the edit distance. Neither of these two distance functions can be meaningfully used in this application. The following example illustrates the point. Consider three strings I242, I233, and H242. These strings are labels (*predicate logic for knowledge representation, logic programming,* and *distributed database systems*) of three fourth-level nodes in the ACM CR classification tree. Nodes I242 and I233 are the grandchildren of the node labeled I2 (*artificial intelligence*) and H242 is a grandchild of the node labeled H2 (*database management*). So, the distance between I242 and I233 should be smaller than that between I242 and H242. However, Hamming distance and edit distance [Baeza-Yates 1992] both have a value 2 between I242 and I233 and a value of 1 between I242 and H242. This limitation motivates the definition of a new similarity measure that correctly captures the similarity between the above strings. The similarity between two strings is defined as the ratio of the length of the largest common prefix [Murty and Jain 1995] between the two strings to the length of the first string. For example, the similarity between strings I522 and I51 is 0.5. The proposed similarity measure is not symmetric because the similarity between I51 and I522 is 0.67. The minimum and maximum values of this similarity measure are 0.0 and 1.0, respectively. The knowledge of the relationship between nodes in the ACM CR classification tree is captured by the representation in the form of strings. For example, node labeled *pattern recognition* is represented by the string I5, whereas the string I53 corresponds to the node labeled *clustering*. The similarity between these two nodes (I5 and I53) is 1.0. A symmetric measure of similarity [Murty and Jain 1995] is used to construct a similarity matrix of size 100 x 100 corresponding to 100 books used in experiments.

6.3.3 *An Algorithm for Clustering Books.* The clustering problem can be stated as follows. Given a collection $\mathcal{B}$ of books, we need to obtain a set $\mathcal{C}$ of clusters. A proximity dendrogram [Jain and Dubes 1988], using the complete-link agglomerative clustering algorithm for the collection of 100 books is shown in Figure 33. Seven clusters are obtained by choosing a threshold (τ) value of 0.12. It is well known that different values for τ might give different clusterings. This threshold value is chosen because the "gap" in the dendrogram between the levels at which six and seven clusters are formed is the largest. An examination of the subject areas of the books [Murty and Jain 1995] in these clusters revealed that the clusters obtained are indeed meaningful. Each of these clusters are represented using a list of string s and frequency s_f pairs, where s_f is the number of books in the cluster in which s is present. For example, cluster $\mathbf{c}_1$ contains 43 books belonging to *pattern recognition, neural networks, artificial intelligence,* and *computer vision*; a part of its representation $\mathcal{R}(C_1)$ is given below.

$$\begin{aligned}\mathcal{R}(C_1) = {} & ((B718,1), (C12,1), (D0,2),\\ & (D311,1), (D312,2), (D321,1),\\ & (D322,1), (D329,1), \ldots (I46,3),\\ & (I461,2), (I462,1), (I463, 3),\\ & \ldots (J26,1), (J6,1),\\ & (J61,7), (J71,1))\end{aligned}$$

These clusters of books and the corresponding cluster descriptions can be used as follows: If a user is searching for books, say, on *image segmentation* (I46), then we select cluster C_1 because its representation alone contains the string I46. Books B_2 (*Neurocomputing*) and B_{18} (*Sensory Neural Networks: Lateral Inhibition*) are both members of cluster C_1 even though their LCC numbers are quite different (B_2 is QA76.5.H4442, B_{18} is $QP363.3.N33$).

Four additional books labeled B_{101}, B_{102}, B_{103}, and B_{104} have been used to study the problem of assigning classification numbers to new books. The LCC numbers of these books are: (B_{101}) Q335.T39, (B_{102}) QA76.73.P356C57, (B_{103}) QA76.5.B76C.2, and (B_{104}) QA76.9D5W44. These books are assigned to clusters based on nearest neighbor classification. The nearest neighbor of B_{101}, a book on *artificial intelligence*, is B_{23} and so B_{101} is assigned to cluster C_1. It is observed that the assignment of these four books to the respective clusters is meaningful, demonstrating that knowledge-based clustering is useful in solving problems associated with document retrieval.

6.4 Data Mining

In recent years we have seen ever increasing volumes of collected data of all sorts. With so much data available, it is necessary to develop algorithms which can extract *meaningful* information from the vast stores. Searching for useful nuggets of information among huge amounts of data has become known as the field of *data mining*.

Data mining can be applied to relational, transaction, and spatial databases, as well as large stores of unstructured data such as the World Wide Web. There are many data mining systems in use today, and applications include the U.S. Treasury detecting money laundering, National Basketball Association coaches detecting trends and patterns of play for individual players and teams, and categorizing patterns of children in the foster care system [Hedberg 1996]. Several journals have had recent special issues on data mining [Cohen 1996, Cross 1996, Wah 1996].

6.4.1 *Data Mining Approaches.* Data mining, like clustering, is an exploratory activity, so clustering methods are well suited for data mining. Clustering is often an important initial step of several in the data mining process [Fayyad 1996]. Some of the data mining approaches which use clustering are *database segmentation*, *predictive modeling*, and *visualization* of large databases.

Segmentation. Clustering methods are used in data mining to segment databases into homogeneous groups. This can serve purposes of data compression (working with the clusters rather than individual items), or to identify characteristics of subpopulations which can be targeted for specific purposes (e.g., marketing aimed at senior citizens).

A *continuous k-means* clustering algorithm [Faber 1994] has been used to cluster pixels in Landsat images [Faber et al. 1994]. Each pixel originally has 7 values from different satellite bands, including infra-red. These 7 values are difficult for humans to assimilate and analyze without assistance. Pixels with the 7 feature values are clustered into 256 groups, then each pixel is assigned the value of the cluster centroid. The image can then be displayed with the spatial information intact. Human viewers can look at a single picture and identify a region of interest (e.g., highway or forest) and label it as a concept. The system then identifies other pixels in the same cluster as an instance of that concept.

Predictive Modeling. Statistical methods of data analysis usually involve hypothesis testing of a model the analyst already has in mind. Data mining can aid the user in discovering potential

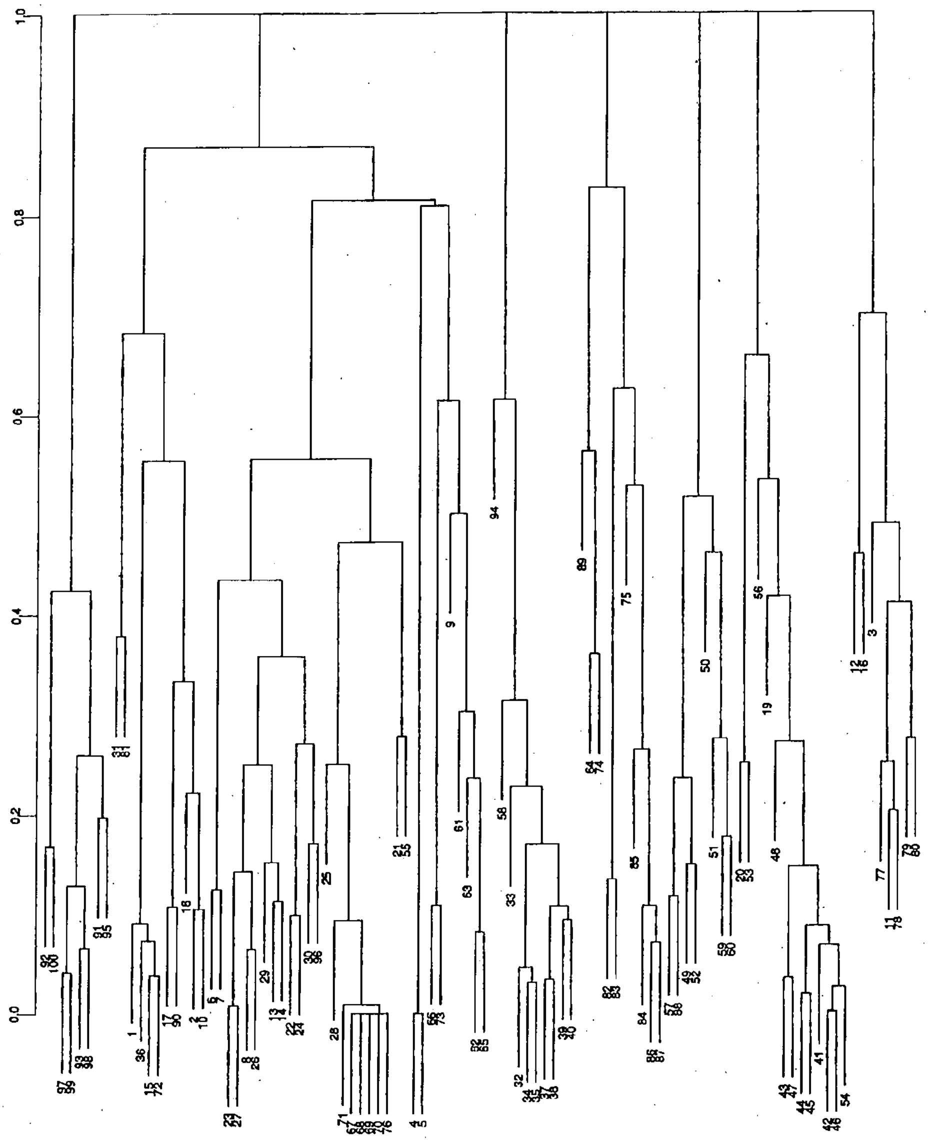


Figure 33. A dendrogram corresponding to 100 books.

hypotheses prior to using statistical tools. Predictive modeling uses clustering to group items, then infers rules to characterize the groups and suggest models. For example, magazine subscribers can be clustered based on a number of factors (age, sex, income, etc.), then the resulting groups characterized in an attempt to find a model which will distinguish those subscribers that will renew their subscriptions from those that will not [Simoudis 1996].

Visualization. Clusters in large databases can be used for visualization, in order to aid human analysts in identifying groups and subgroups that have similar characteristics. WinViz [Lee and Ong 1996] is a data mining visualization

Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6	Cluster 7
employe	applic	action	cadmaz	cfr	amend	anim
fmla	claim	affirm	consult	contain	bankruptci	commod
leav	file	american	copyright	cosmet	code	cpg
	invent	discrimin	custom	ey	court	except
	patent	job	design	hair	creditor	fat
	provision	minor	manag	ingredi	debtor	fe
		opportun	project	label	petition	food
		peopl	sect	manufactur	properti	fruit
		women	servic	product	section	level
				regul	secur	ppm
					truste	refer
						top
						veget

Figure 34. The seven smallest clusters found in the document set. These are stemmed words.

tool in which derived clusters can be exported as new attributes which can then be characterized by the system. For example, breakfast cereals are clustered according to calories, protein, fat, sodium, fiber, carbohydrate, sugar, potassium, and vitamin content per serving. Upon seeing the resulting clusters, the user can export the clusters to Win-Viz as attributes. The system shows that one of the clusters is characterized by high potassium content, and the human analyst recognizes the individuals in the cluster as belonging to the "bran" cereal family, leading to a generalization that "bran cereals are high in potassium."

6.4.2 *Mining Large Unstructured Databases.* Data mining has often been performed on transaction and relational databases which have well-defined fields which can be used as features, but there has been recent research on large unstructured databases such as the World Wide Web [Etzioni 1996].

Examples of recent attempts to classify Web documents using words or functions of words as features include Maarek and Shaul [1996] and Chekuri et al. [1999]. However, relatively small sets of labeled training samples and very large dimensionality limit the ultimate success of automatic Web document categorization based on words as features.

Rather than grouping documents in a word feature space, Wulfekuhler and Punch [1997] cluster the words from a small collection of World Wide Web documents in the document space. The sample data set consisted of 85 documents from the manufacturing domain in 4 different user-defined categories (*labor, legal, government, and design*). These 85 documents contained 5190 distinct word stems after common words (the, and, of) were removed. Since the words are certainly not uncorrelated, they should fall into clusters where words used in a consistent way across the document set have similar values of frequency in each document.

K-means clustering was used to group the 5190 words into 10 groups. One surprising result was that an average of 92% of the words fell into a single cluster, which could then be discarded for data mining purposes. The smallest clusters contained terms which to a human seem semantically related. The 7 smallest clusters from a typical run are shown in Figure 34.

Terms which are used in ordinary contexts, or unique terms which do not occur often across the training document set will tend to cluster into the large 4000 member group. This takes care of spelling errors, proper names which are infrequent, and terms which are used in the same manner throughout the entire document set. Terms used in specific contexts (such as *file* in the context of filing a patent, rather than a computer file) will appear in the documents consistently with other terms appropriate to that context (*patent, invent*) and thus will tend to cluster together. Among the groups of words, unique contexts stand out from the crowd.

After discarding the largest cluster, the smaller set of features can be used to construct queries for seeking out other relevant documents on the Web using standard Web searching tools (e.g., Lycos, Alta Vista, Open Text).

Searching the Web with terms taken from the word clusters allows discovery of finer grained topics (*e.g.*, family medical leave) within the broadly defined categories (e.g., labor).

6.4.3 *Data Mining in Geological Databases.* Database mining is a critical resource in oil exploration and production. It is common knowledge in the oil industry that the typical cost of drilling a new offshore well is in the range of \$30-40 million, but the chance of that site being an economic success is 1 in 10. More informed and systematic drilling decisions can significantly reduce overall production costs.

Advances in drilling technology and data collection methods have led to oil companies and their ancillaries collecting large amounts of geophysical/geological data from production wells and exploration sites, and then organizing them into large databases. Data mining techniques has recently been used to derive precise analytic relations between observed phenomena and parameters. These relations can then be used to quantify oil and gas reserves.

In qualitative terms, good recoverable reserves have high hydrocarbon saturation that are trapped by highly porous sediments (reservoir porosity) and surrounded by hard bulk rocks that prevent the hydrocarbon from leaking away. A large volume of porous sediments is crucial to finding good recoverable reserves, therefore developing reliable and accurate methods for estimation of sediment porosities from the collected data is key to estimating hydrocarbon potential.

The general rule of thumb experts use for porosity computation is that it is a quasiexponential function of depth:

$$Porosity = K \cdot e^{-F(x_1, x_2, \ldots, x_m) \cdot Depth}. \qquad (4)$$

A number of factors such as rock types, structure, and cementation as parameters of function F confound this relationship. This necessitates the definition of proper *contexts*, in which to attempt discovery of porosity formulas. Geological contexts are expressed in terms of geological phenomena, such as geometry, lithology, compaction, and subsidence, associated with a region. It is well known that geological context changes from basin to basin (different geographical areas in the world) and also from region to region within a basin [Allen and Allen 1990; Biswas 1995]. Furthermore, the underlying features of contexts may vary greatly. Simple model matching techniques, which work in engineering domains where behavior is constrained by man-made systems and well-established laws of physics, may not apply in the hydrocarbon exploration domain. To address this, data clustering was used to identify the relevant contexts, and then equation discovery was carried out within each context. The goal was to derive the subset x_1, x_2, ..., x_m from a larger set of geological features, and the functional relationship F that best defined the porosity function in a region.

The overall methodology illustrated in Figure 35, consists of two primary steps: (i) Context definition using unsupervised clustering techniques, and (ii) Equation discovery by regression analysis [Li and Biswas 1995]. Real exploration data collected from a region in the

(1) **Context Definition**

1.1 discover *primitive structures* $(g_1, g_2, \ldots, g_m)$ by clustering,

1.2 define *context* in terms of the relevant sequences of primitive structures, i.e., $C_i = g_{i1} \circ g_{i2} \circ, \ldots, \circ g_{ik}$,

1.3 group data according to the context definition to form *homogeneous data groups*,

1.4 for each relevant data group, determine the set of *relevant variables* $(x_1, x_2, \ldots, x_k)$ for porosity.

(2) **Equation Derivation**

2.1 select possible *base models* (equations) using domain theory,

2.2 use the *least squares* method to generate coefficient values for each base model,

2.3 use the *component plus residual plot* (*cprp*) heuristic to dynamically modify the equation model to better fit the data,

Figure 35. Description of the knowledge-based scientific discovery process.

Alaska basin was analyzed using the methodology developed. The data objects (patterns) are described in terms of 37 geological features, such as porosity, permeability, grain size, density, and sorting, amount of different mineral fragments (e.g., quartz, chert, feldspar) present, nature of the rock fragments, pore characteristics, and cementation. All these feature values are numeric measurements made on samples obtained from well-logs during exploratory drilling processes.

The k-means clustering algorithm was used to identify a set of homogeneous primitive geological structures (g_1, g_2, ..., g_m). These primitives were then mapped onto the unit code versus stratigraphic unit map. Figure 36 depicts a partial mapping for a set of wells and four primitive structures. The next step in the discovery process identified sections of wells regions that were made up of the same sequence of geological primitives. Every sequence defined a context C_i. From the partial mapping of Figure 36, the context $C_1 = g_2 \circ g_1 \circ g_2 \circ g_3$ was identified in two well regions (the 300 and 600 series). After the contexts were defined, data points belonging to each context were grouped together for equation derivation. The derivation procedure employed multiple regression analysis [Sen and Srivastava 1990].

This method was applied to a data set of about 2600 objects corresponding to sample measurements collected from wells is the Alaskan Basin. The k-means clustered this data set into seven groups. As an illustration, we selected a set of 138 objects representing a context for further analysis. The features that best defined this cluster were selected, and experts surmised that the context represented a low porosity region, which was modeled using the regression procedure.

7. SUMMARY

There are several applications where decision making and exploratory pattern analysis have to be performed on large data sets. For example, in document retrieval, a set of relevant documents has to be found among several millions of documents of dimensionality of more than 1000. It is possible to handle these problems if some useful abstraction of the data is obtained and is used in decision making, rather than directly using the entire data set. By *data abstraction*, we mean a simple and compact representation of the data. This simplicity helps the machine in efficient processing or a human in comprehending the structure in data easily. Clustering algorithms are ideally suited for achieving data abstraction.

In this paper, we have examined various steps in clustering: (1) pattern representation, (2) similarity computation, (3) grouping process, and (4) cluster representation. Also, we have discussed

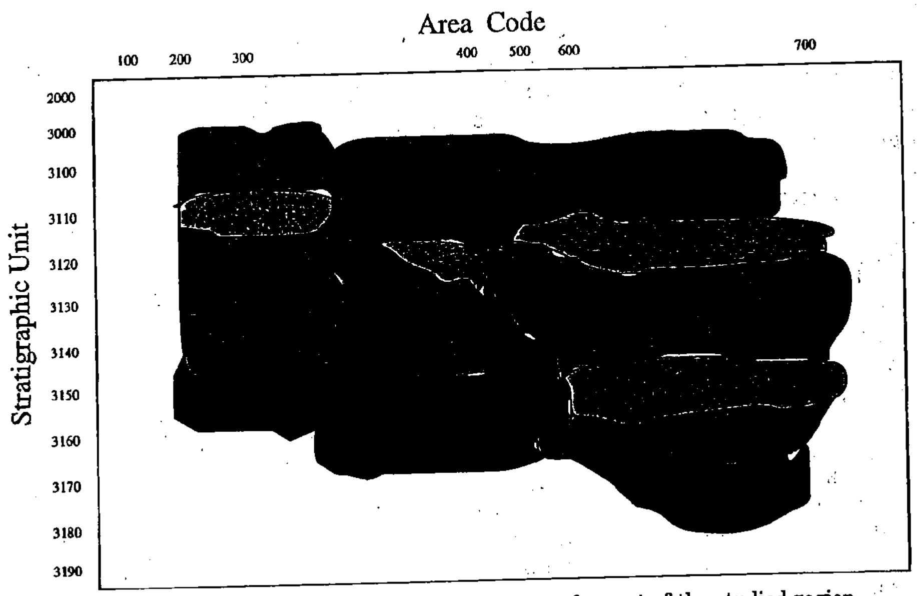


Figure 36. Area code versus stratigraphic unit map for part of the studied region.

statistical, fuzzy, neural, evolutionary, and knowledge-based approaches to clustering. We have described four applications of clustering: (1) image segmentation, (2) object recognition, (3) document retrieval, and (4) data mining.

Clustering is a process of grouping data items based on a measure of similarity. Clustering is a subjective process; the same set of data items often needs to be partitioned differently for different applications. This subjectivity makes the process of clustering difficult. This is because a single algorithm or approach is not adequate to solve every clustering problem. A possible solution lies in reflecting this subjectivity in the form of knowledge. This knowledge is used either implicitly or explicitly in one or more phases of clustering. Knowledge-based clustering algorithms use domain knowledge explicitly.

The most challenging step in clustering is feature extraction or pattern representation. Pattern recognition researchers conveniently avoid this step by assuming that the pattern representations are available as input to the clustering algorithm. In small size data sets, pattern representations can be obtained based on previous experience of the user with the problem. However, in the case of large data sets, it is difficult for the user to keep track of the importance of each feature in clustering. A solution is to make as many measurements on the patterns as possible and use them in pattern representation. But it is not possible to use a large collection of measurements directly in clustering because of computational costs. So several feature extraction/selection approaches have been designed to obtain linear or nonlinear combinations of these measurements which can be used to represent patterns. Most of the schemes proposed for feature extraction/selection are typically iterative in nature and cannot be used on large data sets due to prohibitive computational costs.

The second step in clustering is similarity computation. A variety of schemes have been used to compute similarity between two patterns. They use knowledge either implicitly or explicitly. Most of the knowledge-based clustering algorithms use explicit knowledge in similarity computation. However, if patterns are not represented using proper features, then it is not possible to get a meaningful partition irrespective of the quality and quantity of knowledge used in similarity computation. There is no universally acceptable scheme for computing similarity between patterns represented using a mixture of both qualitative and quantitative features. Dissimilarity between a pair of patterns is represented using a distance measure that may or may not be a metric.

The next step in clustering is the grouping step. There are broadly two grouping schemes: hierarchical and partitional schemes. The hierarchical schemes are more versatile, and the partitional schemes are less expensive. The partitional algorithms aim at maximizing the squared error criterion function. Motivated by the failure of the squared error partitional clustering algorithms in finding the optimal solution to this problem, a large collection of approaches have been proposed and used to obtain the global optimal solution to this problem. However, these schemes are computationally prohibitive on large data sets. ANN-based clustering schemes are neural implementations of the clustering algorithms, and they share the undesired properties of these algorithms. However, ANNs have the capability to automatically normalize the data and extract features. An important observation is that even if a scheme can find the optimal solution to the squared error partitioning problem, it may still fall short of the requirements because of the possible non-isotropic nature of the clusters.

In some applications, for example in document retrieval, it may be useful to have a clustering that is not a partition. This means clusters are overlapping. Fuzzy clustering and functional clustering are ideally suited for this purpose. Also, fuzzy clustering algorithms can handle mixed data types. However, a major problem with fuzzy clustering is that it is difficult to obtain the membership values. A general approach may not work because of the subjective nature of clustering. It is required to represent clusters obtained in a suitable form to help the decision maker. Knowledge-based clustering schemes generate intuitively appealing descriptions of clusters. They can be used even when the patterns are represented using a combination of qualitative and quantitative features, provided that knowledge linking a concept and the mixed features are available. However, implementations of the conceptual clustering schemes are computationally expensive and are not suitable for grouping large data sets.

The k-means algorithm and its neural implementation, the Kohonen net, are most successfully used on large data sets. This is because k-means algorithm is simple to implement and computationally attractive because of its linear time complexity. However, it is not feasible to use even this linear time algorithm on large data sets. Incremental algorithms like leader and its neural implementation, the ART network, can be used to cluster large data sets. But they tend to be order-dependent. *Divide and conquer* is a heuristic that has been rightly exploited by computer algorithm designers to reduce computational costs. However, it should be judiciously used in clustering to achieve meaningful results.

In summary, clustering is an interesting, useful, and challenging problem. It has great potential in applications like object recognition, image segmentation, and information filtering and retrieval. However, it is possible to exploit this potential only after making several design choices carefully.

ACKNOWLEDGMENTS

The authors wish to acknowledge the generosity of several colleagues who read manuscript drafts, made suggestions, and provided summaries of emerging application areas which we have incorporated into this paper. Gautam Biswas and Cen Li of Vanderbilt University provided the material on knowledge discovery in geological databases. Ana Fred of Instituto Superior Técnico in Lisbon, Portugal provided material on cluster analysis in the syntactic domain. William Punch and Marilyn Wulfekuhler of Michigan State University provided material on the application of cluster analysis to data mining problems. Scott Connell of Michigan State provided material describing his work on character recognition. Chitra Dorai of IBM T.J. Watson Research Center provided material on the use of clustering in 3D object recognition. Jianchang Mao of IBM Almaden Research Center, Peter Bajcsy of the University of Illinois, and Zoran Obradović of Washington State University also provided many helpful comments. Mario de Figueirido performed a meticulous reading of the manuscript and provided many helpful suggestions.

This work was supported by the National Science Foundation under grant INT-9321584.

REFERENCES

Aarts, E. and Korst, J. 1989. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley and Sons, Inc., New York, NY.

ACM, 1994. ACM CR Classifications. *ACM Computing Surveys 35*, 5–16.

Al-Sultan, K. S. 1995. A tabu search approach to clustering problems. *Pattern Recogn. 28*, 1443–1451.

Al-Sultan, K. S. and Khan, M. M. 1996. Computational experience on four algorithms for the hard clustering problem. *Pattern Recogn. Lett. 17*, 3, 295–308.

Allen, P. A. and Allen, J. R. 1990. *Basin Analysis: Principles and Applications*. Blackwell Scientific Publications, Inc., Cambridge, MA.

Alta Vista, 1999. http://altavista.digital.com.

Amadasun, M. and King, R. A. 1988. Low-level segmentation of multispectral images via agglomerative clustering of uniform neighbourhoods. *Pattern Recogn. 21*, 3 (1988), 261–268.

Anderberg, M. R. 1973. *Cluster Analysis for Applications*. Academic Press, Inc., New York, NY.

Augustson, J. G. and Minker, J. 1970. An analysis of some graph theoretical clustering techniques. *J. ACM 17*, 4 (Oct. 1970), 571–588.

Babu, G. P. and Murty, M. N. 1993. A near-optimal initial seed value selection in *K*-means algorithm using a genetic algorithm. *Pattern Recogn. Lett. 14*, 10 (Oct. 1993), 763–769.

Babu, G. P. and Murty, M. N. 1994. Clustering with evolution strategies. *Pattern Recogn. 27*, 321–329.

Babu, G. P., Murty, M. N., and Keerthi, S. S. 2000. Stochastic connectionist approach for pattern clustering (To appear). *IEEE Trans. Syst. Man Cybern.*.

Backer, F. B. and Hubert, L. J. 1976. A graph-theoretic approach to goodness-of-fit in complete-link hierarchical clustering. *J. Am. Stat. Assoc. 71*, 870–878.

Backer, E. 1995. *Computer-Assisted Reasoning in Cluster Analysis*. Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Baeza-Yates, R. A. 1992. Introduction to data structures and algorithms related to information retrieval. In *Information Retrieval: Data Structures and Algorithms*, W. B. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, Inc., Upper Saddle River, NJ, 13–27.

Bajcsy, P. 1997. Hierarchical segmentation and clustering using similarity analysis. Ph.D. Dissertation. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

Ball, G. H. and Hall, D. J. 1965. ISODATA, a novel method of data analysis and classification. Tech. Rep.. Stanford University, Stanford, CA.

Bentley, J. L. and Friedman, J. H. 1978. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Trans. Comput. C-27*, 6 (June), 97–105.

Bezdek, J. C. 1981. *Pattern Recognition With Fuzzy Objective Function Algorithms*. Plenum Press, New York, NY.

Bhuyan, J. N., Raghavan, V. V., and Venkatesh, K. E. 1991. Genetic algorithm for clustering with an ordered representation. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 408–415.

Biswas, G., Weinberg, J., and Li, C. 1995. *A Conceptual Clustering Method for Knowledge Discovery in Databases*. Editions Technip.

Brailovsky, V. L. 1991. A probabilistic approach to clustering. *Pattern Recogn. Lett. 12*, 4 (Apr. 1991), 193–198.

Brodatz, P. 1966. *Textures: A Photographic Album for Artists and Designers*. Dover Publications, Inc., Mineola, NY.

Can, F. 1993. Incremental clustering for dynamic information processing. *ACM Trans. Inf. Syst. 11*, 2 (Apr. 1993), 143–164.

Carpenter, G. and Grossberg, S. 1990. ART3: Hierarchical search using chemical transmitters in self-organizing pattern recognition architectures. *Neural Networks 3*, 129–152.

Chekuri, C., Goldwasser, M. H., Raghavan, P., and Upfal, E. 1997. Web search using automatic classification. In *Proceedings of the Sixth International Conference on the World Wide Web* (Santa Clara, CA, Apr.), http://theory.stanford.edu/people/wass/publications/Web Search/Web Search.html.

Cheng, C. H. 1995. A branch-and-bound clustering algorithm. *IEEE Trans. Syst. Man Cybern. 25*, 895–898.

Cheng, Y. and Fu, K. S. 1985. Conceptual clustering in knowledge organization. *IEEE Trans. Pattern Anal. Mach. Intell. 7*, 592–598.

Cheng, Y. 1995. Mean shift, mode seeking, and clustering. *IEEE Trans. Pattern Anal. Mach. Intell. 17*, 7 (July), 790–799.

Chien, Y. T. 1978. *Interactive Pattern Recognition*. Marcel Dekker, Inc., New York, NY.

Choudhury, S. and Murty, M. N. 1990. A divisive scheme for constructing minimal spanning trees in coordinate space. *Pattern Recogn. Lett. 11*, 6 (Jun. 1990), 385–389.

1996. Special issue on data mining. *Commun. ACM 39*, 11.

Coleman, G. B. and Andrews, H. C. 1979. Image segmentation by clustering. *Proc. IEEE 67*, 5, 773–785.

Connell, S. and Jain, A. K. 1998. Learning prototypes for on-line handwritten digits. In *Proceedings of the 14th International Conference on Pattern Recognition* (Brisbane, Australia, Aug.), 182–184.

Cross, S. E., Ed. 1996. Special issue on data mining. *IEEE Expert 11*, 5 (Oct.).

Dale, M. B. 1985. On the comparison of conceptual clustering and numerical taxonomy. *IEEE Trans. Pattern Anal. Mach. Intell. 7*, 241–244.

Dave, R. N. 1992. Generalized fuzzy C-shells clustering and detection of circular and elliptic boundaries. *Pattern Recogn. 25*, 713–722.

Davis, T., Ed. 1991. *The Handbook of Genetic Algorithms*. Van Nostrand Reinhold Co., New York, NY.

Day, W. H. E. 1992. Complexity theory: An introduction for practitioners of classification. In *Clustering and Classification*, P. Arabie and L. Hubert, Eds. World Scientific Publishing Co., Inc., River Edge, NJ.

Dempster, A. P., Laird, N. M., and Rubin, D. B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Stat. Soc. B. 39*, 1, 1–38.

Diday, E. 1973. The dynamic cluster method in non-hierarchical clustering. *J. Comput. Inf. Sci. 2*, 61–88.

Diday, E. and Simon, J. C. 1976. Clustering analysis. In *Digital Pattern Recognition*, K. S. Fu, Ed. Springer-Verlag, Secaucus, NJ, 47–94.

Diday, E. 1988. The symbolic approach in clustering. In *Classification and Related Methods*, H. H. Bock, Ed. North-Holland Publishing Co., Amsterdam, The Netherlands.

Dorai, C. and Jain, A. K. 1995. Shape spectra based view grouping for free-form objects. In *Proceedings of the International Conference on Image Processing* (ICIP-95), 240–243.

Dubes, R. C. and Jain, A. K. 1976. Clustering techniques: The user's dilemma. *Pattern Recogn. 8*, 247–260.

Dubes, R. C. and Jain, A. K. 1980. Clustering methodology in exploratory data analysis. In *Advances in Computers*, M. C. Yovits,, Ed. Academic Press, Inc., New York, NY, 113–125.

Dubes, R. C. 1987. How many clusters are best?—an experiment. *Pattern Recogn. 20*, 6 (Nov. 1, 1987), 645–663.

Dubes, R. C. 1993. Cluster analysis and related issues. In *Handbook of Pattern Recognition & Computer Vision*, C. H. Chen, L. F. Pau, and P. S. P. Wang, Eds. World Scientific Publishing Co., Inc., River Edge, NJ, 3–32.

Dubuisson, M. P. and Jain, A. K. 1994. A modified Hausdorff distance for object matchin-g. In *Proceedings of the International Conference on Pattern Recognition* (ICPR '94), 566–568.

Duda, R. O. and Hart, P. E. 1973. *Pattern Classification and Scene Analysis*. John Wiley and Sons, Inc., New York, NY.

Dunn, S., Janos, L., and Rosenfeld, A. 1983. Bimean clustering. *Pattern Recogn. Lett. 1*, 169–173.

Duran, B. S. and Odell, P. L. 1974. *Cluster Analysis: A Survey*. Springer-Verlag, New York, NY.

Eddy, W. F., Mockus, A., and Oue, S. 1996. Approximate single linkage cluster analysis of large data sets in high-dimensional spaces. *Comput. Stat. Data Anal. 23*, 1, 29–43.

Etzioni, O. 1996. The World-Wide Web: quagmire or gold mine? *Commun. ACM 39*, 11, 65–68.

Everitt, B. S. 1993. *Cluster Analysis*. Edward Arnold, Ltd., London, UK.

Faber, V. 1994. Clustering and the continuous k-means algorithm. *Los Alamos Science 22*, 138–144.

Faber, V., Hochberg, J. C., Kelly, P. M., Thomas, T. R., and White, J. M. 1994. Concept extraction: A data-mining technique. *Los Alamos Science 22*, 122–149.

Fayyad, U. M. 1996. Data mining and knowledge discovery: Making sense out of data. *IEEE Expert 11*, 5 (Oct.), 20–25.

FISHER, D. AND LANGLEY, P. 1986. Conceptual clustering and its relation to numerical taxonomy. In *Artificial Intelligence and Statistics*, A W. Gale, Ed. Addison-Wesley Longman Publ. Co., Inc., Reading, MA, 77–116.

FISHER, D. 1987. Knowledge acquisition via incremental conceptual clustering. *Mach. Learn. 2*, 139–172.

FISHER, D., XU, L., CARNES, R., RICH, Y., FENVES, S. J., CHEN, J., SHIAVI, R., BISWAS, G., AND WEINBERG, J. 1993. Applying AI clustering to engineering tasks. *IEEE Expert 8*, 51–60.

FISHER, L. AND VAN NESS, J. W. 1971. Admissible clustering procedures. *Biometrika 58*, 91–104.

FLYNN, P. J. AND JAIN, A. K. 1991. BONSAI: 3D object recognition using constrained search. *IEEE Trans. Pattern Anal. Mach. Intell. 13*, 10 (Oct. 1991), 1066–1075.

FOGEL, D. B. AND SIMPSON, P. K. 1993. Evolving fuzzy clusters. In *Proceedings of the International Conference on Neural Networks* (San Francisco, CA), 1829–1834.

FOGEL, D. B. AND FOGEL, L. J., Eds. 1994. Special issue on evolutionary computation. *IEEE Trans. Neural Netw.* (Jan.).

FOGEL, L. J., OWENS, A. J., AND WALSH, M. J. 1965. *Artificial Intelligence Through Simulated Evolution*. John Wiley and Sons, Inc., New York, NY.

FRAKES, W. B. AND BAEZA-YATES, R., Eds. 1992. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ.

FRED, A. L. N. AND LEITAO, J. M. N. 1996. A minimum code length technique for clustering of syntactic patterns. In *Proceedings of the International Conference on Pattern Recognition* (Vienna, Austria), 680–684.

FRED, A. L. N. 1996. Clustering of sequences using a minimum grammar complexity criterion. In *Grammatical Inference: Learning Syntax from Sentences*, L. Miclet and C. Higuera, Eds. Springer-Verlag, Secaucus, NJ, 107–116.

FU, K. S. AND LU, S. Y. 1977. A clustering procedure for syntactic patterns. *IEEE Trans. Syst. Man Cybern. 7*, 734–742.

FU, K. S. AND MUI, J. K. 1981. A survey on image segmentation. *Pattern Recogn. 13*, 3–16.

FUKUNAGA, K. 1990. *Introduction to Statistical Pattern Recognition*. 2nd ed. Academic Press Prof., Inc., San Diego, CA.

GLOVER, F. 1986. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res. 13*, 5 (May 1986), 533–549.

GOLDBERG, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Co., Inc., Redwood City, CA.

GORDON, A. D. AND HENDERSON, J. T. 1977. Algorithm for Euclidean sum of squares. *Biometrics 33*, 355–362.

GOTLIEB, G. C. AND KUMAR, S. 1968. Semantic clustering of index terms. *J. ACM 15*, 493–513.

GOWDA, K. C. 1984. A feature reduction and unsupervised classification algorithm for multispectral data. *Pattern Recogn. 17*, 6, 667–676.

GOWDA, K. C. AND KRISHNA, G. 1977. Agglomerative clustering using the concept of mutual nearest neighborhood. *Pattern Recogn. 10*, 105–112.

GOWDA, K. C. AND DIDAY, E. 1992. Symbolic clustering using a new dissimilarity measure. *IEEE Trans. Syst. Man Cybern. 22*, 368–378.

GOWER, J. C. AND ROSS, G. J. S. 1969. Minimum spanning rees and single-linkage cluster analysis. *Appl. Stat. 18*, 54–64.

GREFENSTETTE, J 1986. Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern. SMC-16*, 1 (Jan./Feb. 1986), 122–128.

HARALICK, R. M. AND KELLY, G. L. 1969. Pattern recognition with measurement space and spatial clustering for multiple images. *Proc. IEEE 57*, 4, 654–665.

HARTIGAN, J. A. 1975. *Clustering Algorithms*. John Wiley and Sons, Inc., New York, NY.

HEDBERG, S. 1996. Searching for the mother lode: Tales of the first data miners. *IEEE Expert 11*, 5 (Oct.), 4–7.

HERTZ, J., KROGH, A., AND PALMER, R. G. 1991. *Introduction to the Theory of Neural Computation*. Santa Fe Institute Studies in the Sciences of Complexity lecture notes. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

HOFFMAN, R. AND JAIN, A. K. 1987. Segmentation and classification of range images. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-9*, 5 (Sept. 1987), 608–620.

HOFMANN, T. AND BUHMANN, J. 1997. Pairwise data clustering by deterministic annealing. *IEEE Trans. Pattern Anal. Mach. Intell. 19*, 1 (Jan.), 1–14.

HOFMANN, T., PUZICHA, J., AND BUCHMANN, J. M. 1998. Unsupervised texture segmentation in a deterministic annealing framework. *IEEE Trans. Pattern Anal. Mach. Intell. 20*, 8, 803–818.

HOLLAND, J. H. 1975. *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

HOOVER, A., JEAN-BAPTISTE, G., JIANG, X., FLYNN, P. J., BUNKE, H., GOLDGOF, D. B., BOWYER, K., EGGERT, D. W., FITZGIBBON, A., AND FISHER, R. B. 1996. An experimental comparison of range image segmentation algorithms. *IEEE Trans. Pattern Anal. Mach. Intell. 18*, 7, 673–689.

HUTTENLOCHER, D. P., KLANDERMAN, G. A., AND RUCKLIDGE, W. J. 1993. Comparing images using the Hausdorff distance. *IEEE Trans. Pattern Anal. Mach. Intell. 15*, 9, 850–863.

ICHINO, M. AND YAGUCHI, H. 1994. Generalized Minkowski metrics for mixed feature-type data analysis. *IEEE Trans. Syst. Man Cybern. 24*, 698–708.

1991. *Proceedings of the International Joint Conference on Neural Networks*. (IJCNN'91).

1992. *Proceedings of the International Joint Conference on Neural Networks*.

ISMAIL, M. A. AND KAMEL, M. S. 1989. Multidimensional data clustering utilizing hybrid search strategies. *Pattern Recogn. 22*, 1 (Jan. 1989), 75–89.

JAIN, A. K. AND DUBES, R. C. 1988. *Algorithms for Clustering Data*. Prentice-Hall advanced reference series. Prentice-Hall, Inc., Upper Saddle River, NJ.

JAIN, A. K. AND FARROKHNIA, F. 1991. Unsupervised texture segmentation using Gabor filters. *Pattern Recogn. 24*, 12 (Dec. 1991), 1167–1186.

JAIN, A. K. AND BHATTACHARJEE, S. 1992. Text segmentation using Gabor filters for automatic document processing. *Mach. Vision Appl. 5*, 3 (Summer 1992), 169–184.

JAIN, A. J. AND FLYNN, P. J., Eds. 1993. *Three Dimensional Object Recognition Systems*. Elsevier Science Inc., New York, NY.

JAIN, A. K. AND MAO, J. 1994. Neural networks and pattern recognition. In *Computational Intelligence: Imitating Life*, J. M. Zurada, R. J. Marks, and C. J. Robinson, Eds. 194–212.

JAIN, A. K. AND FLYNN, P. J. 1996. Image segmentation using clustering. In *Advances in Image Understanding: A Festschrift for Azriel Rosenfeld*, N. Ahuja and K. Bowyer, Eds, IEEE Press, Piscataway, NJ, 65–83.

JAIN, A. K. AND MAO, J. 1996. Artificial neural networks: A tutorial. *IEEE Computer 29* (Mar.), 31–44.

JAIN, A. K., RATHA, N. K., AND LAKSHMANAN, S. 1997. Object detection using Gabor filters. *Pattern Recogn. 30*, 2, 295–309.

JAIN, N. C., INDRAYAN, A., AND GOEL, L. R. 1986. Monte Carlo comparison of six hierarchical clustering methods on random data. *Pattern Recogn. 19*, 1 (Jan./Feb. 1986), 95–99.

JAIN, R., KASTURI, R., AND SCHUNCK, B. G. 1995. *Machine Vision*. McGraw-Hill series in computer science. McGraw-Hill, Inc., New York, NY.

JARVIS, R. A. AND PATRICK, E. A. 1973. Clustering using a similarity method based on shared near neighbors. *IEEE Trans. Comput. C-22*, 8 (Aug.), 1025–1034.

JOLION, J.-M., MEER, P., AND BATAOUCHE, S. 1991. Robust clustering with applications in computer vision. *IEEE Trans. Pattern Anal. Mach. Intell. 13*, 8 (Aug. 1991), 791–802.

JONES, D. AND BELTRAMO, M. A. 1991. Solving partitioning problems with genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 442–449.

JUDD, D., MCKINLEY, P., AND JAIN, A. K. 1996. Large-scale parallel data clustering. In *Proceedings of the International Conference on Pattern Recognition* (Vienna, Austria), 488–493.

KING, B. 1967. Step-wise clustering procedures. *J. Am. Stat. Assoc. 69*, 86–101.

KIRKPATRICK, S., GELATT, C. D., JR., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598 (May), 671–680.

KLEIN, R. W. AND DUBES, R. C. 1989. Experiments in projection and clustering by simulated annealing. *Pattern Recogn. 22*, 213–220.

KNUTH, D. 1973. *The Art of Computer Programming*. Addison-Wesley, Reading, MA.

KOONTZ, W. L. G., FUKUNAGA, K., AND NARENDRA, P. M. 1975. A branch and bound clustering algorithm. *IEEE Trans. Comput. 23*, 908–914.

KOHONEN, T. 1989. *Self-Organization and Associative Memory*. 3rd ed. Springer information sciences series. Springer-Verlag, New York, NY.

KRAAIJVELD, M., MAO, J., AND JAIN, A. K. 1995. A non-linear projection method based on Kohonen's topology preserving maps. *IEEE Trans. Neural Netw. 6*, 548–559.

KRISHNAPURAM, R., FRIGUI, H., AND NASRAOUI, O. 1995. Fuzzy and probabilistic shell clustering algorithms and their application to boundary detection and surface approximation. *IEEE Trans. Fuzzy Systems 3*, 29–60.

KURITA, T. 1991. An efficient agglomerative clustering algorithm using a heap. *Pattern Recogn. 24*, 3 (1991), 205–209.

LIBRARY OF CONGRESS, 1990. LC classification outline. Library of Congress, Washington, DC.

LEBOWITZ, M. 1987. Experiments with incremental concept formation. *Mach. Learn. 2*, 103–138.

LEE, H.-Y. AND ONG, H.-L. 1996. Visualization support for data mining. *IEEE Expert 11*, 5 (Oct.), 69–75.

LEE, R. C. T., SLAGLE, J. R., AND MONG, C. T. 1978. Towards automatic auditing of records. *IEEE Trans. Softw. Eng. 4*, 441–448.

LEE, R. C. T. 1981. Cluster analysis and its applications. In *Advances in Information Systems Science*, J. T. Tou, Ed. Plenum Press, New York, NY.

LI, C. AND BISWAS, G. 1995. Knowledge-based scientific discovery in geological databases. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining* (Montreal, Canada, Aug. 20-21), 204–209.

LU, S. Y. AND FU, K. S. 1978. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Trans. Syst. Man Cybern. 8,* 381–389.

LUNDERVOLD, A., FENSTAD, A. M., ERSLAND, L., AND TAXT, T. 1996. Brain tissue volumes from multispectral 3D MRI: A comparative study of four classifiers. In *Proceedings of the Conference of the Society on Magnetic Resonance,*

MAAREK, Y. S. AND BEN SHAUL, I. Z. 1996. Automatically organizing bookmarks per contents. In *Proceedings of the Fifth International Conference on the World Wide Web* (Paris, May), http://www5conf.inria.fr/fich-html/paper-sessions.html.

MCQUEEN, J. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability,* 281–297.

MAO, J. AND JAIN, A. K. 1992. Texture classification and segmentation using multiresolution simultaneous autoregressive models. *Pattern Recogn. 25,* 2 (Feb. 1992), 173–188.

MAO, J. AND JAIN, A. K. 1995. Artificial neural networks for feature extraction and multivariate data projection. *IEEE Trans. Neural Netw. 6,* 296–317.

MAO, J. AND JAIN, A. K. 1996. A self-organizing network for hyperellipsoidal clustering (HEC). *IEEE Trans. Neural Netw. 7,* 16–29.

MEVINS, A. J. 1995. A branch and bound incremental conceptual clusterer. *Mach. Learn. 18,* 5–22.

MICHALSKI, R., STEPP, R. E., AND DIDAY, E. 1981. A recent advance in data analysis: Clustering objects into classes characterized by conjunctive concepts. In *Progress in Pattern Recognition, Vol. 1,* L. Kanal and A. Rosenfeld, Eds. North-Holland Publishing Co., Amsterdam, The Netherlands.

MICHALSKI, R., STEPP, R. E., AND DIDAY, E. 1983. Automated construction of classifications: conceptual clustering versus numerical taxonomy. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-5,* 5 (Sept.), 396–409.

MISHRA, S. K. AND RAGHAVAN, V. V. 1994. An empirical study of the performance of heuristic methods for clustering. In *Pattern Recognition in Practice,* E. S. Gelsema and L. N. Kanal, Eds. 425–436.

MITCHELL,T. 1997. *MachineLearning.* McGraw-Hill, Inc., New York, NY.

MOHIUDDIN, K. M. AND MAO, J. 1994. A comparative study of different classifiers for handprinted character recognition. In *Pattern Recognition in Practice,* E. S. Gelsema and L. N. Kanal, Eds. 437–448.

MOOR, B. K. 1988. ART 1 and Pattern Clustering. In *1988 Connectionist Summer School,* Morgan Kaufmann, San Mateo, CA, 174–185.

MURTAGH, F. 1984. A survey of recent advances in hierarchical clustering algorithms which use cluster centers. *Comput. J. 26,* 354–359.

MURTY, M. N. AND KRISHNA, G. 1980. A computationally efficient technique for data clustering. *Pattern Recogn. 12,* 153–158.

MURTY, M. N. AND JAIN, A. K. 1995. Knowledge-based clustering scheme for collection management and retrieval of library books. *Pattern Recogn. 28,* 949–964.

NAGY, G. 1968. State of the art in pattern recognition. *Proc. IEEE 56,* 836–862.

NG, R. AND HAN, J. 1994. Very large data bases. In *Proceedings of the 20th International Conference on Very Large Data Bases* (VLDB'94, Santiago, Chile, Sept.), VLDB Endowment, Berkeley, CA, 144–155.

NGUYEN, H. H. AND COHEN, P. 1993. Gibbs random fields, fuzzy clustering, and the unsupervised segmentation of textured images. *CVGIP: Graph. Models Image Process. 55,* 1 (Jan. 1993), 1–19.

OEHLER, K. L. AND GRAY, R. M. 1995. Combining image compression and classification using vector quantization. *IEEE Trans. Pattern Anal. Mach. Intell. 17,* 461–473.

OJA, E. 1982. A simplified neuron model as a principal component analyzer. *Bull. Math. Bio. 15,* 267–273.

OZAWA, K. 1985. A stratificational overlapping cluster scheme. *Pattern Recogn. 18,* 279–286.

OPEN TEXT, 1999. http://index.opentext.net.

KAMGAR-PARSI, B., GUALTIERI, J. A., DEVANEY, J. A., AND KAMGAR-PARSI, K. 1990. Clustering with neural networks. *Biol. Cybern. 63,* 201–208.

LYCOS, 1999. http://www.lycos.com.

PAL, N. R., BEZDEK, J. C., AND TSAO, E. C.-K. 1993. Generalized clustering networks and Kohonen's self-organizing scheme. *IEEE Trans. Neural Netw. 4,* 549–557.

QUINLAN, J. R. 1990. Decision trees and decision making. *IEEE Trans. Syst. Man Cybern. 20,* 339–346.

RAGHAVAN, V. V. AND BIRCHAND, K. 1979. A clustering strategy based on a formalism of the reproductive process in a natural system. In *Proceedings of the Second International Conference on Information Storage and Retrieval,* 10–22.

RAGHAVAN, V. V. AND YU, C. T. 1981. A comparison of the stability characteristics of some graph theoretic clustering methods. *IEEE Trans. Pattern Anal. Mach. Intell. 3,* 393–402.

RASMUSSEN, E. 1992. Clustering algorithms. In *Information Retrieval: Data Structures and Algorithms,* W. B. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, Inc., Upper Saddle River, NJ, 419–442.

RICH, E. 1983. *ArtificialIntelligence.* McGraw-Hill, Inc., New York, NY.

RIPLEY, B. D., Ed. 1989. *Statistical Inference for Spatial Processes.* Cambridge University Press, New York, NY.

ROSE, K., GUREWITZ, E., AND FOX, G. C. 1993. Deterministic annealing approach to constrained clustering. *IEEE Trans. Pattern Anal. Mach. Intell. 15,* 785–794.

Rosenfeld, A. and Kak, A. C. 1982. *Digital Picture Processing*. 2nd ed. Academic Press, Inc., New York, NY.

Rosenfeld, A., Schneider, V. B., and Huang, M. K. 1969. An application of cluster detection to text and picture processing. *IEEE Trans. Inf. Theor. 15*, 6, 672–681.

Ross, G. J. S. 1968. Classification techniques for large sets of data. In *Numerical Taxonomy*, A. J. Cole, Ed. Academic Press, Inc., New York, NY.

Ruspini, E. H. 1969. A new approach to clustering. *Inf. Control 15*, 22–32.

Salton, G. 1991. Developments in automatic text retrieval. *Science 253*, 974–980.

Samal, A. and Iyengar, P. A. 1992. Automatic recognition and analysis of human faces and facial expressions: A survey. *Pattern Recogn. 25*, 1 (Jan. 1992), 65–77.

Sammon, J. W. Jr. 1969. A nonlinear mapping for data structure analysis. *IEEE Trans. Comput. 18*, 401–409.

Sangal, R. 1991. *Programming Paradigms in LISP*. McGraw-Hill, Inc., New York, NY.

Schachter, B. J., Davis, L. S., and Rosenfeld, A. 1979. Some experiments in image segmentation by clustering of local feature values. *Pattern Recogn. 11*, 19–28.

Schwefel, H. P. 1981. *Numerical Optimization of Computer Models*. John Wiley and Sons, Inc., New York, NY.

Selim, S. Z. and Ismail, M. A. 1984. K-means-type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Trans. Pattern Anal. Mach. Intell. 6*, 81–87.

Selim, S. Z. and Alsultan, K. 1991. A simulated annealing algorithm for the clustering problem. *Pattern Recogn. 24*, 10 (1991), 1003–1008.

Sen, A. and Srivastava, M. 1990. *Regression Analysis*. Springer-Verlag, New York, NY.

Sethi, I. and Jain, A. K., Eds. 1991. *Artificial Neural Networks and Pattern Recognition: Old and New Connections*. Elsevier Science Inc., New York, NY.

Shekar, B., Murty, N. M., and Krishna, G. 1987. A knowledge-based clustering scheme. *Pattern Recogn. Lett. 5*, 4 (Apr. 1, 1987), 253–259.

Silverman, J. F. and Cooper, D. B. 1988. Bayesian clustering for unsupervised estimation of surface and texture models. *IEEE Trans. Pattern Anal. Mach. Intell. 10*, 4 (July 1988), 482–495.

Simoudis, E. 1996. Reality check for data mining. *IEEE Expert 11*, 5 (Oct.), 26–33.

Slagle, J. R., Chang, C. L., and Heller, S. R. 1975. A clustering and data-reorganizing algorithm. *IEEE Trans. Syst. Man Cybern. 5*, 125–128.

Sneath, P. H. A. and Sokal, R. R. 1973. *Numerical Taxonomy*. Freeman, London, UK.

Spath, H. 1980. *Cluster Analysis Algorithms for Data Reduction and Classification*. Ellis Horwood, Upper Saddle River, NJ.

Solberg, A., Taxt, T., and Jain, A. 1996. A Markov random field model for classification of multisource satellite imagery. *IEEE Trans. Geoscience and Remote Sensing 34*, 1, 100–113.

Srivastava, A. and Murty, M. N 1990. A comparison between conceptual clustering and conventional clustering. *Pattern Recogn. 23*, 9 (1990), 975–981.

Stahl, H. 1986. Cluster analysis of large data sets. In *Classification as a Tool of Research*, W. Gaul and M. Schader, Eds. Elsevier North-Holland, Inc., New York, NY, 423–430.

Stepp, R. E. and Michalski, R. S. 1986. Conceptual clustering of structured objects: A goal-oriented approach. *Artif. Intell. 28*, 1 (Feb. 1986), 43–69.

Sutton, M., Stark, L., and Bowyer, K. 1993. Function-based generic recognition for multiple object categories. In *Three-Dimensional Object Recognition Systems*, A. Jain and P. J. Flynn, Eds. Elsevier Science Inc., New York, NY.

Symon, M. J. 1977. Clustering criterion and multi-variate normal mixture. *Biometrics 77*, 35–43.

Tanaka, E. 1995. Theoretical aspects of syntactic pattern recognition. *Pattern Recogn. 28*, 1053–1061.

Taxt, T. and Lundervold, A. 1994. Multispectral analysis of the brain using magnetic resonance imaging. *IEEE Trans. Medical Imaging 13*, 3, 470–481.

Titterington, D. M., Smith, A. F. M., and Makov, U. E. 1985. *Statistical Analysis of Finite Mixture Distributions*. John Wiley and Sons, Inc., New York, NY.

Toussaint, G. T. 1980. The relative neighborhood graph of a finite planar set. *Pattern Recogn. 12*, 261–268.

Trier, O. D. and Jain, A. K. 1995. Goal-directed evaluation of binarization methods. *IEEE Trans. Pattern Anal. Mach. Intell. 17*, 1191–1201.

Uchiyama, T. and Arbib, M. A. 1994. Color image segmentation using competitive learning. *IEEE Trans. Pattern Anal. Mach. Intell. 16*, 12 (Dec. 1994), 1197–1206.

Urquhart, R. B. 1982. Graph theoretical clustering based on limited neighborhood sets. *Pattern Recogn. 15*, 173–187.

Venkateswarlu, N. B. and Raju, P. S. V. S. K. 1992. Fast ISODATA clustering algorithms. *Pattern Recogn. 25*, 3 (Mar. 1992), 335–342.

Vinod, V. V., Chaudhury, S., Mukherjee, J., and Ghose, S. 1994. A connectionist approach for clustering with applications in image analysis. *IEEE Trans. Syst. Man Cybern. 24*, 365–384.

WAH, B. W., Ed. 1996. Special section on mining of databases. *IEEE Trans. Knowl. Data Eng.* (Dec.).

WARD, J. H. JR. 1963. Hierarchical grouping to optimize an objective function. *J. Am. Stat. Assoc. 58*, 236–244.

WATANABE, S. 1985. *Pattern Recognition: Human and Mechanical.* John Wiley and Sons, Inc., New York, NY.

WESZKA, J. 1978. A survey of threshold selection techniques. *Pattern Recogn. 7*, 259–265.

WHITLEY, D., STARKWEATHER, T., AND FUQUAY, D. 1989. Scheduling problems and traveling salesman: the genetic edge recombination. In *Proceedings of the Third International Conference on Genetic Algorithms* (George Mason University, June 4–7), J. D. Schaffer, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 133–140.

WILSON, D. R. AND MARTINEZ, T. R. 1997. Improved heterogeneous distance functions. *J. Artif. Intell. Res. 6*, 1–34.

WU, Z. AND LEAHY, R. 1993. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell. 15*, 1101–1113.

WULFEKUHLER, M. AND PUNCH, W. 1997. Finding salient features for personal web page categories. In *Proceedings of the Sixth International Conference on the World Wide Web* (Santa Clara, CA, Apr.), http://theory.stanford.edu/people/wass/publications/Web Search/Web Search.html.

ZADEH, L. A. 1965. Fuzzy sets. *Inf. Control 8*, 338–353.

ZAHN, C. T. 1971. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Comput. C-20* (Apr.), 68–86.

ZHANG, K. 1995. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recogn. 28*, 463–474.

ZHANG, J. AND MICHALSKI, R. S. 1995. An integration of rule induction and exemplar-based learning for graded concepts. *Mach. Learn. 21*, 3 (Dec. 1995), 235–267.

ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. 1996. BIRCH: An efficient data clustering method for very large databases. *SIGMOD Rec. 25*, 2, 103–114.

ZUPAN, J. 1982. *Clustering of Large Data Sets.* Research Studies Press Ltd., Taunton, UK.

Received: March 1997; revised: October 1998; accepted: January 1999

EXHIBIT C

Chapter 6

UNSUPERVISED LEARNING AND CLUSTERING

6.1 INTRODUCTION

Until now we have assumed that the training samples used to design a classifier were labelled to show their category membership. Procedures that use labelled samples are said to be supervised. Now we shall investigate a number of *unsupervised* procedures that use unlabelled samples. That is, we shall see what can be done when all one has is a collection of samples without being told their classification.

One might wonder why anyone is interested in such an unpromising problem, and whether or not it is even possible in principle to learn anything of value from unlabelled samples. There are three basic reasons for interest in unsupervised procedures. First, the collection and labelling of a large set of sample patterns can be surprisingly costly and time consuming. If a classifier can be crudely designed on a small, labelled set of samples, and then "tuned up" by allowing it to run without supervision on a large, unlabelled set, much time and trouble can be saved. Second, in many applications the characteristics of the patterns can change slowly with time. If these changes can be tracked by a classifier running in an unsupervised mode, improved performance can be achieved. Finally, in the early stages of an investigation it may be valuable to gain some insight into the nature or structure of the data. The discovery of distinct subclasses or major departures from expected characteristics may significantly alter the approach taken to designing the classifier.

The answer to the question of whether or not it is possible in principle to learn anything from unlabelled data depends upon the assumptions one is willing to accept—theorems can not be proved without premises. We shall begin with the very restrictive assumption that the functional forms for the underlying probability densities are known, and that the only thing that must be learned is the value of an unknown parameter vector. Interestingly enough, the formal solution to this problem will turn out to be almost identical to the solution for the problem of supervised learning given in Chapter 3. Unfortunately, in the unsupervised case the solution suffers from the usual problems associated with parametric assumptions without providing any of the benefits of computational simplicity. This will lead us to various attempts to reformulate the problem as one of partitioning the data into subgroups or clusters. While some of the resulting clustering procedures have no known significant theoretical properties, they are still among the more useful tools for pattern recognition problems.

6.2 MIXTURE DENSITIES AND IDENTIFIABILITY

We begin by assuming that we know the complete probability structure for the problem with the sole exception of the values of some parameters. To be more specific, we make the following assumptions:

(1) The samples come from a known number c of classes.
(2) The a priori probabilities $P(\omega_j)$ for each class are known, $j = 1, \ldots, c$.
(3) The forms for the class-conditional probability densities $p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j)$ are known, $j = 1, \ldots, c$.
(4) All that is unknown are the values for the c parameter vectors $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_c$.

Samples are assumed to be obtained by selecting a state of nature ω_j with probability $P(\omega_j)$ and then selecting an $\mathbf{x}$ according to the probability law $p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j)$. Thus, the probability density function for the samples is given by

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j) P(\omega_j), \tag{1}$$

where $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_c)$. A density function of this form is called a *mixture density*. The conditional densities $p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j)$ are called the *component densities*, and the a priori probabilities $P(\omega_j)$ are called the *mixing parameters*. The mixing parameters can also be included among the unknown parameters, but for the moment we shall assume that only $\boldsymbol{\theta}$ is unknown.

Our basic goal will be to use samples drawn from this mixture density to estimate the unknown parameter vector $\boldsymbol{\theta}$. Once we know $\boldsymbol{\theta}$ we can decompose the mixture into its components, and the problem is solved. Before seeking explicit solutions to this problem, however, let us ask whether or not it is possible in principle to recover $\boldsymbol{\theta}$ from the mixture. Suppose that we had an unlimited number of samples, and that we used one of the nonparametric methods of Chapter 4 to determine the value of $p(\mathbf{x} \mid \boldsymbol{\theta})$ for every $\mathbf{x}$. If there is only one value of $\boldsymbol{\theta}$ that will produce the observed values for $p(\mathbf{x} \mid \boldsymbol{\theta})$, then a solution is at least possible in principle. However, if several different values of $\boldsymbol{\theta}$ can produce the same values for $p(\mathbf{x} \mid \boldsymbol{\theta})$, then there is no hope of obtaining a unique solution.

These considerations lead us to the following definition: a density $p(\mathbf{x} \mid \boldsymbol{\theta})$ is said to be *identifiable* if $\boldsymbol{\theta} \neq \boldsymbol{\theta}'$ implies that there exists an $\mathbf{x}$ such that $p(\mathbf{x} \mid \boldsymbol{\theta}) \neq p(\mathbf{x} \mid \boldsymbol{\theta}')$. As one might expect, the study of unsupervised learning is greatly simplified if we restrict ourselves to identifiable mixtures. Fortunately, most mixtures of commonly encountered density functions are identifiable. Mixtures of discrete distributions are not always so obliging. For a simple example, consider the case where x is binary and $P(x \mid \boldsymbol{\theta})$ is the mixture

$$\begin{aligned} P(x \mid \boldsymbol{\theta}) &= \tfrac{1}{2}\theta_1^x(1 - \theta_1)^{1-x} + \tfrac{1}{2}\theta_2^x(1 - \theta_2)^{1-x} \\ &= \begin{cases} \tfrac{1}{2}(\theta_1 + \theta_2) & \text{if } x = 1 \\ 1 - \tfrac{1}{2}(\theta_1 + \theta_2) & \text{if } x = 0. \end{cases} \end{aligned}$$

If we know, for example, that $P(x = 1 \mid \boldsymbol{\theta}) = 0.6$, and hence that $P(x = 0 \mid \boldsymbol{\theta}) = 0.4$, then we know the function $P(x \mid \boldsymbol{\theta})$, but we cannot determine $\boldsymbol{\theta}$, and hence cannot extract the component distributions. The most we can say is that $\theta_1 + \theta_2 = 1.2$. Thus, here we have a case in which the mixture distribution is not identifiable, and hence a case for which unsupervised learning is impossible in principle.

This kind of problem commonly occurs with discrete distributions. If there are too many components in the mixture, there may be more unknowns than independent equations, and identifiability can be a real problem. For the continuous case, the problems are less severe, although certain minor difficulties can arise due to the possibility of special cases. Thus, while it can be shown that mixtures of normal densities are usually identifiable, the parameters in the simple mixture density

$$p(x \mid \boldsymbol{\theta}) = \frac{P(\omega_1)}{\sqrt{2\pi}} \exp[-\tfrac{1}{2}(x - \theta_1)^2] + \frac{P(\omega_2)}{\sqrt{2\pi}} \exp[-\tfrac{1}{2}(x - \theta_2)^2]$$

can not be uniquely identified if $P(\omega_1) = P(\omega_2)$, for then θ_1 and θ_2 can be interchanged without affecting $p(x \mid \boldsymbol{\theta})$. To avoid such irritations, we shall acknowledge that identifiability can be a problem, but shall henceforth assume that the mixture densities we are working with are identifiable.

6.3 MAXIMUM LIKELIHOOD ESTIMATES

Suppose now that we are given a set $\mathscr{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of n unlabelled samples drawn independently from the mixture density

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j), \tag{1}$$

where the parameter vector $\boldsymbol{\theta}$ is fixed but unknown. The likelihood of the observed samples is by definition the joint density

$$p(\mathscr{X} \mid \boldsymbol{\theta}) = \prod_{k=1}^{n} p(\mathbf{x}_k \mid \boldsymbol{\theta}). \tag{2}$$

The maximum likelihood estimate $\hat{\boldsymbol{\theta}}$ is that value of $\boldsymbol{\theta}$ that maximizes $p(\mathscr{X} \mid \boldsymbol{\theta})$.

If we assume that $p(\mathscr{X} \mid \boldsymbol{\theta})$ is a differentiable function of $\boldsymbol{\theta}$, then we can derive some interesting necessary conditions for $\hat{\boldsymbol{\theta}}$. Let l be the logarithm of the likelihood, and let $\nabla_{\boldsymbol{\theta}_i} l$ be the gradient of l with respect to $\boldsymbol{\theta}_i$. Then

$$l = \sum_{k=1}^{n} \log p(\mathbf{x}_k \mid \boldsymbol{\theta}) \tag{3}$$

and

$$\nabla_{\boldsymbol{\theta}_i} l = \sum_{k=1}^{n} \frac{1}{p(\mathbf{x}_k \mid \boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}_i}\left[\sum_{j=1}^{c} p(\mathbf{x}_k \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j)\right].$$

If we assume that the elements of $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_j$ are functionally independent if $i \neq j$, and if we introduce the a posteriori probability

$$P(\omega_i \mid \mathbf{x}_k, \boldsymbol{\theta}) = \frac{p(\mathbf{x}_k \mid \omega_i, \boldsymbol{\theta}_i)P(\omega_i)}{p(\mathbf{x}_k \mid \boldsymbol{\theta})} \tag{4}$$

we see that the gradient of the log-likelihood can be written in the interesting form

$$\nabla_{\boldsymbol{\theta}_i} l = \sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}_i} \log p(\mathbf{x}_k \mid \omega_i, \boldsymbol{\theta}_i). \tag{5}$$

Since the gradient must vanish at the $\boldsymbol{\theta}_i$ that maximizes l, the maximum-likelihood estimate $\hat{\boldsymbol{\theta}}_i$ must satisfy the conditions

$$\sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \nabla_{\boldsymbol{\theta}_i} \log p(\mathbf{x}_k \mid \omega_i, \hat{\boldsymbol{\theta}}_i) = 0, \qquad i = 1, \ldots, c. \tag{6}$$

Conversely, among the solutions to these equations for $\hat{\boldsymbol{\theta}}_i$ we will find the maximum-likelihood solution.

It is not hard to generalize these results to include the a priori probabilities $P(\omega_i)$ among the unknown quantities. In this case the search for the maximum value of $p(\mathscr{X} \mid \boldsymbol{\theta})$ extends over $\boldsymbol{\theta}$ and $P(\omega_i)$, subject to the constraints

$$P(\omega_i) \geq 0 \qquad i = 1, \ldots, c \tag{7}$$

and

$$\sum_{i=1}^{c} P(\omega_i) = 1. \tag{8}$$

Let $\hat{P}(\omega_i)$ be the maximum likelihood estimate for $P(\omega_i)$, and let $\hat{\boldsymbol{\theta}}_i$ be the maximum likelihood estimate for $\boldsymbol{\theta}_i$. The diligent reader will be able to show that if the likelihood function is differentiable and if $\hat{P}(\omega_i) \neq 0$ for any i, then $\hat{P}(\omega_i)$ and $\hat{\boldsymbol{\theta}}_i$ must satisfy

$$\hat{P}(\omega_i) = \frac{1}{n} \sum_{k=1}^{n} \hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \tag{9}$$

and

$$\sum_{k=1}^{n} \hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \nabla_{\boldsymbol{\theta}_i} \log p(\mathbf{x}_k \mid \omega_i, \hat{\boldsymbol{\theta}}_i) = 0, \tag{10}$$

where

$$\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) = \frac{p(\mathbf{x}_k \mid \omega_i, \hat{\boldsymbol{\theta}}_i)\hat{P}(\omega_i)}{\sum_{j=1}^{c} p(\mathbf{x}_k \mid \omega_j, \hat{\boldsymbol{\theta}}_j)\hat{P}(\omega_j)}. \tag{11}$$

6.4 APPLICATION TO NORMAL MIXTURES

It is enlightening to see how these general results apply to the case where the component densities are multivariate normal, $p(\mathbf{x} \mid \omega_i, \boldsymbol{\theta}_i) \sim N(\boldsymbol{\mu}_i, \Sigma_i)$. The following table illustrates a few of the different cases that can arise depending upon which parameters are known (√) and which are unknown (?):

Case	$\boldsymbol{\mu}_i$	Σ_i	$P(\omega_i)$	c
1	?	√	√	√
2	?	?	?	√
3	?	?	?	?

Case 1 is the simplest, and will be considered in detail because of its pedagogic value. Case 2 is more realistic, though somewhat more involved.

Case 3 represents the problem we face on encountering a completely unknown set of data. Unfortunately, it can not be solved by maximum-likelihood methods. We shall postpone discussion of what can be done when the number of classes is unknown until later in this chapter.

6.4.1 Case 1: Unknown Mean Vectors

If the only unknown quantities are the mean vectors $\boldsymbol{\mu}_i$, then $\boldsymbol{\theta}_i$ can be identified with $\boldsymbol{\mu}_i$ and Eq. (6) can be used to obtain necessary conditions on the maximum likelihood estimate for $\boldsymbol{\mu}_i$. Since

$$\log p(\mathbf{x} \mid \omega_i, \boldsymbol{\mu}_i) = -\log[(2\pi)^{d/2}|\Sigma_i|^{1/2}] - \tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^t\Sigma_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i),$$

$$\nabla_{\boldsymbol{\mu}_i} \log p(\mathbf{x} \mid \omega_i, \boldsymbol{\mu}_i) = \Sigma_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i).$$

Thus, Eq. (6) for the maximum-likelihood estimate $\hat{\boldsymbol{\mu}}_i$ yields

$$\sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}})\Sigma_i^{-1}(\mathbf{x}_k - \hat{\boldsymbol{\mu}}_i) = 0, \text{ where } \hat{\boldsymbol{\mu}} = (\hat{\boldsymbol{\mu}}_1, \ldots, \hat{\boldsymbol{\mu}}_c).$$

After multiplying by Σ_i and rearranging terms, we obtain

$$\hat{\boldsymbol{\mu}}_i = \frac{\sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}})\mathbf{x}_k}{\sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}})}. \tag{12}$$

This equation is intuitively very satisfying. It shows that the estimate for $\boldsymbol{\mu}_i$ is merely a weighted average of the samples. The weight for the kth sample is an estimate of how likely it is that $\mathbf{x}_k$ belongs to the ith class. If $P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}})$ happened to be one for some of the samples and zero for the rest, then $\hat{\boldsymbol{\mu}}_i$ would be the mean of those samples estimated to belong to the ith class. More generally, suppose that $\hat{\boldsymbol{\mu}}_i$ is sufficiently close to the true value of $\boldsymbol{\mu}_i$ that $P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}})$ is essentially the true a posteriori probability for ω_i. If we think of $P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}})$ as the fraction of those samples having value $\mathbf{x}_k$ that come from the ith class, then we see that Eq. (12) essentially gives $\hat{\boldsymbol{\mu}}_i$ as the average of the samples coming from the ith class.

Unfortunately, Eq. (12) does not give $\hat{\boldsymbol{\mu}}_i$ explicitly, and if we substitute

$$P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}}) = \frac{p(\mathbf{x}_k \mid \omega_i, \hat{\boldsymbol{\mu}}_i)P(\omega_i)}{\sum_{j=1}^{c} p(\mathbf{x}_k \mid \omega_j, \hat{\boldsymbol{\mu}}_j)P(\omega_j)}$$

with $p(\mathbf{x} \mid \omega_i, \hat{\boldsymbol{\mu}}_i) \sim N(\hat{\boldsymbol{\mu}}_i, \Sigma_i)$, we obtain a tangled snarl of coupled simultaneous nonlinear equations. These equations usually do not have a unique solution, and we must test the solutions we get to find the one that actually maximizes the likelihood.

If we have some way of obtaining fairly good initial estimates $\hat{\boldsymbol{\mu}}_i(0)$ for the unknown means, Eq. (12) suggests the following iterative scheme for improving the estimates:

$$\hat{\boldsymbol{\mu}}_i(j+1) = \frac{\sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}}(j))\mathbf{x}_k}{\sum_{k=1}^{n} P(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\mu}}(j))}. \tag{13}$$

This is basically a gradient ascent or hill-climbing procedure for maximizing the log-likelihood function. If the overlap between component densities is small, then the coupling between classes will be small and convergence will be fast. However, when convergence does occur, all that we can be sure of is that the gradient is zero. Like all hill-climbing procedures, this one carries no guarantee of yielding the global maximum.

6.4.2 An Example

To illustrate the kind of behavior that can occur, consider the simple one-dimensional, two-component normal mixture

$$p(x \mid \mu_1, \mu_2) = \frac{1}{3\sqrt{2\pi}} \exp[-\tfrac{1}{2}(x - \mu_1)^2] + \frac{2}{3\sqrt{2\pi}} \exp[-\tfrac{1}{2}(x - \mu_2)^2].$$

The 25 samples shown in Table 6-1 were drawn from this mixture with

TABLE 6-1. Twenty-five Samples from a Normal Mixture

k	x_k	(Class)
1	0.608	2
2	−1.590	1
3	0.235	2
4	3.949	2
5	−2.249	1
6	2.704	2
7	−2.473	1
8	0.672	2
9	0.262	2
10	1.072	2
11	−1.773	1
12	0.537	2
13	3.240	2
14	2.400	2
15	−2.499	1
16	2.608	2
17	−3.458	1
18	0.257	2
19	2.569	2
20	1.415	2
21	1.410	2
22	−2.653	1
23	1.396	2
24	3.286	2
25	−0.712	1

$\mu_1 = -2$ and $\mu_2 = 2$. Let us use these samples to compute the log-likelihood function

$$l(\mu_1, \mu_2) = \sum_{k=1}^{n} \log p(x_k \mid \mu_1, \mu_2)$$

for various values of μ_1 and μ_2. Figure 6.1 is a contour plot that shows how l varies with μ_1 and μ_2. The maximum value of l occurs at $\hat{\mu}_1 = -2.130$ and $\hat{\mu}_2 = 1.668$, which is in the rough vicinity of the true values $\mu_1 = -2$ and $\mu_2 = 2$.* However, l reaches another peak of comparable height at $\hat{\mu}_1 = 2.085$ and $\hat{\mu}_2 = -1.257$. Roughly speaking, this solution corresponds to interchanging μ_1 and μ_2. Note that had the a priori probabilities been equal,

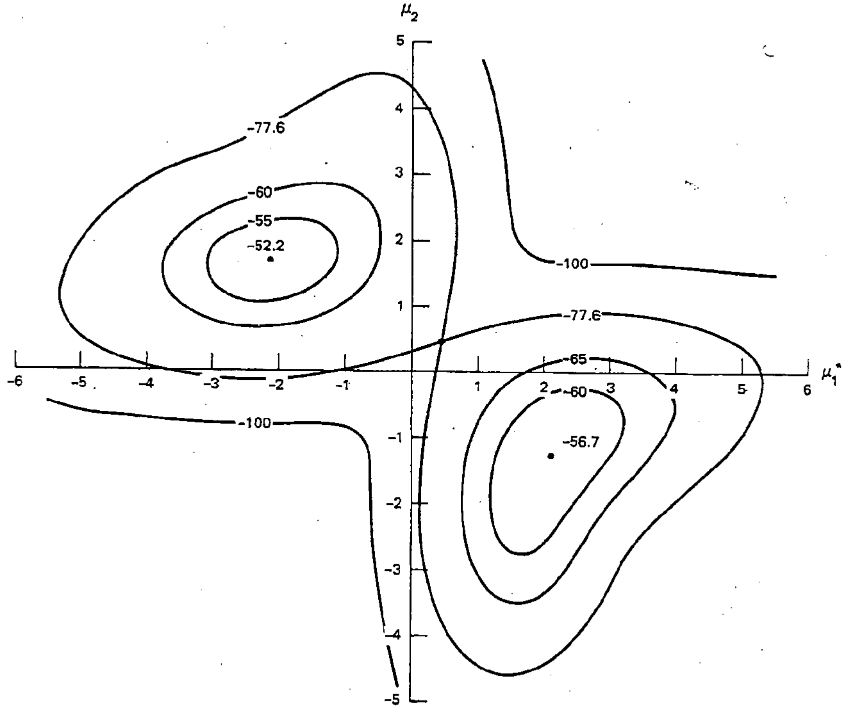


FIGURE 6.1. Contours of a log-likelihood function.

* If the data in Table 6-1 are separated by class, the resulting sample means are $m_1 = -2.176$ and $m_2 = 1.684$. Thus, the maximum likelihood estimates for the unsupervised case are close to the maximum likelihood estimates for the supervised case.

interchanging μ_1 and μ_2 would have produced no change in the log-likelihood function. Thus, when the mixture density is not identifiable, the maximum likelihood solution is not unique.

Additional insight into the nature of these multiple solutions can be obtained by examining the resulting estimates for the mixture density. Figure 6.2 shows the true mixture density and the estimates obtained by using the maximum likelihood estimates as if they were the true parameter values. The 25 sample values are shown as a scatter of points along the abscissa. Note that the peaks of both the true mixture density and the maximum likelihood solution are located so as to encompass two major groups of data points. The estimate corresponding to the smaller local maximum of the log-likelihood function has a mirror-image shape, but its peaks also encompass reasonable groups of data points. To the eye, neither of these solutions is clearly superior, and both are interesting.

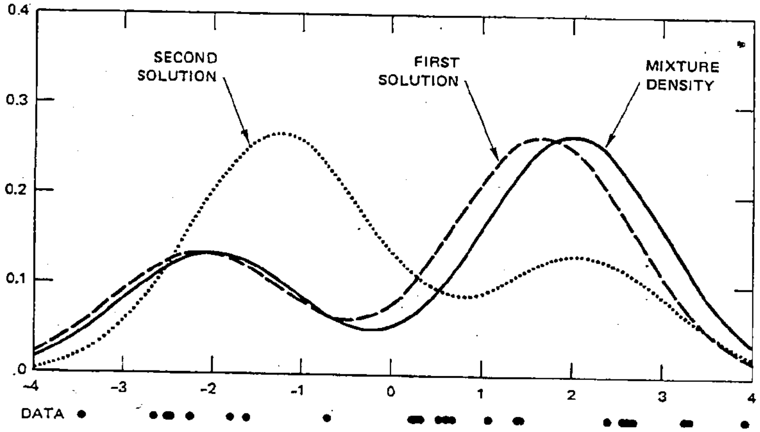


FIGURE 6.2. Estimates of the mixture density.

If Eq. (13) is used to determine solutions to Eq. (12) iteratively, the results depend on the starting values $\hat{\mu}_1(0)$ and $\hat{\mu}_2(0)$. Figure 6.3 shows how different starting points lead to different solutions, and gives some indication of rates of convergence. Note that if $\hat{\mu}_1(0) = \hat{\mu}_2(0)$, convergence to a saddle point occurs in one step. This is not a coincidence. It happens for the simple reason that for this starting point $P(\omega_i \mid x_k, \hat{\mu}_1(0), \hat{\mu}_2(0)) = P(\omega_i)$. Thus, Eq. (13) yields the mean of all of the samples for $\hat{\mu}_1$ and $\hat{\mu}_2$ for all successive iterations. Clearly, this is a general phenomenon, and such saddle-point solutions can

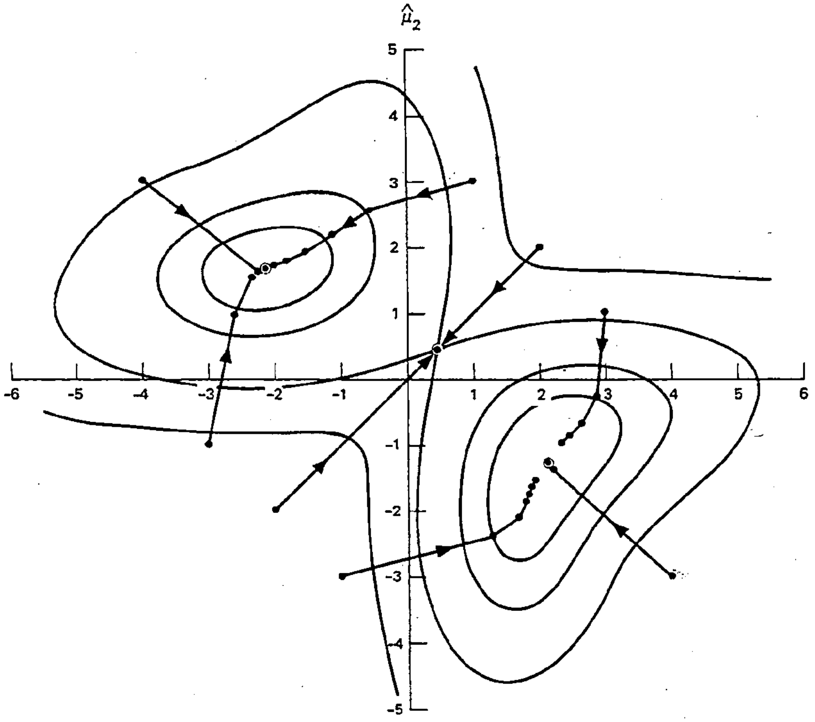


FIGURE 6.3. Trajectories for the iterative procedure.

be expected if the starting point does not bias the search away from a symmetric answer.

6.4.3 Case 2: All Parameters Unknown

If $\boldsymbol{\mu}_i$, Σ_i, and $P(\omega_i)$ are all unknown, and if no constraints are placed on the covariance matrix, then the maximum likelihood principle yields useless singular solutions. The reason for this can be appreciated from the following simple example. Let $p(x \mid \mu, \sigma^2)$ be the two-component normal mixture

$$p(x \mid \mu, \sigma^2) = \frac{1}{2\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right] + \frac{1}{2\sqrt{2\pi}} \exp[-\tfrac{1}{2}x^2].$$

The likelihood function for n samples drawn according to this probability law is merely the product of the n densities $p(x_k \mid \mu, \sigma^2)$. Suppose that we let $\mu = x_1$, so that

$$p(x_1 \mid \mu, \sigma^2) = \frac{1}{2\sqrt{2\pi}\sigma} + \frac{1}{2\sqrt{2\pi}} \exp[-\tfrac{1}{2}x_1^2].$$

Clearly, for the rest of the samples

$$p(x_k \mid \mu, \sigma^2) \geq \frac{1}{2\sqrt{2\pi}} \exp[-\tfrac{1}{2}x_k^2],$$

so that

$$p(x_1, \ldots, x_n \mid \mu, \sigma^2) \geq \left\{\frac{1}{\sigma} + \exp[-\tfrac{1}{2}x_1^2]\right\} \frac{1}{(2\sqrt{2\pi})^n} \exp\left[-\tfrac{1}{2}\sum_{k=2}^{n} x_k^2\right].$$

Thus, by letting σ approach zero we can make the likelihood arbitrarily large, and the maximum likelihood solution is singular.

Ordinarily, singular solutions are of no interest, and we are forced to conclude that the maximum likelihood principle fails for this class of normal mixtures. However, it is an empirical fact that meaningful solutions can still be obtained if we restrict our attention to the largest of the finite local maxima of the likelihood function. Assuming that the likelihood function is well behaved at such maxima, we can use Eqs. (9)–(11) to obtain estimates for $\boldsymbol{\mu}_i$, Σ_i, and $P(\omega_i)$. When we include the elements of Σ_i in the elements of the parameter vector $\boldsymbol{\theta}_i$, we must remember that only half of the off-diagonal elements are independent. In addition, it turns out to be much more convenient to let the independent elements of Σ_i^{-1} rather than Σ_i be the unknown parameters. With these observations, the actual differentiation of

$$\log p(\mathbf{x}_k \mid \omega_i, \boldsymbol{\theta}_i) = \log \frac{|\Sigma_i^{-1}|^{1/2}}{(2\pi)^{d/2}} - \tfrac{1}{2}(\mathbf{x}_k - \boldsymbol{\mu}_i)^t \Sigma_i^{-1}(\mathbf{x}_k - \boldsymbol{\mu}_i)$$

with respect to the elements of $\boldsymbol{\mu}_i$ and Σ_i^{-1} is relatively routine. Let $x_p(k)$ be the pth element of $\mathbf{x}_k$, $\mu_p(i)$ be the pth element of $\boldsymbol{\mu}_i$, $\sigma_{pq}(i)$ be the pqth element of Σ_i, and $\sigma^{pq}(i)$ be the pqth element of Σ_i^{-1}. Then

$$\nabla_{\boldsymbol{\mu}_i} \log p(\mathbf{x}_k \mid \omega_i, \boldsymbol{\theta}_i) = \Sigma_i^{-1}(\mathbf{x}_k - \boldsymbol{\mu}_i)$$

and

$$\frac{\partial \log p(\mathbf{x}_k \mid \omega_i, \boldsymbol{\theta}_i)}{\partial \sigma^{pq}(i)} = \left(1 - \frac{\delta_{pq}}{2}\right)[\sigma_{pq}(i) - (x_p(k) - \mu_p(i))(x_q(k) - \mu_q(i))],$$

where δ_{pq} is the Kronecker delta. Substituting these results in Eq. (10) and doing a small amount of algebraic manipulation, we obtain the following equations for the local-maximum-likelihood estimates $\hat{\mu}_i$, $\hat{\Sigma}_i$, and $\hat{P}(\omega_i)$:

$$\hat{P}(\omega_i) = \frac{1}{n}\sum_{k=1}^{n}\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \tag{14}$$

$$\hat{\mu}_i = \frac{\sum_{k=1}^{n}\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})\mathbf{x}_k}{\sum_{k=1}^{n}\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})} \tag{15}$$

$$\hat{\Sigma}_i = \frac{\sum_{k=1}^{n}\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})(\mathbf{x}_k - \hat{\mu}_i)(\mathbf{x}_k - \hat{\mu}_i)^t}{\sum_{k=1}^{n}\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}_i)} \tag{16}$$

where

$$\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) = \frac{p(\mathbf{x}_k \mid \omega_i, \hat{\boldsymbol{\theta}}_i)\hat{P}(\omega_i)}{\sum_{j=1}^{c} p(\mathbf{x}_k \mid \omega_j, \hat{\boldsymbol{\theta}}_j)\hat{P}(\omega_j)}$$

$$= \frac{|\hat{\Sigma}_i|^{-1/2}\exp[-\frac{1}{2}(\mathbf{x}_k - \hat{\mu}_i)^t\hat{\Sigma}_i^{-1}(\mathbf{x}_k - \hat{\mu}_i)]\hat{P}(\omega_i)}{\sum_{j=1}^{c}|\hat{\Sigma}_j|^{-1/2}\exp[-\frac{1}{2}(\mathbf{x}_k - \hat{\mu}_j)^t\hat{\Sigma}_j^{-1}(\mathbf{x}_k - \hat{\mu}_j)]\hat{P}(\omega_j)}. \tag{17}$$

While the notation may make these equations appear to be rather formidable, their interpretation is actually quite simple. In the extreme case where $\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})$ is one when $\mathbf{x}_k$ is from Class ω_i and zero otherwise, $\hat{P}(\omega_i)$ is the fraction of samples from ω_i, $\hat{\mu}_i$ is the mean of those samples, and $\hat{\Sigma}_i$ is the corresponding sample covariance matrix. More generally, $\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})$ is between zero and one, and all of the samples play some role in the estimates. However, the estimates are basically still frequency ratios, sample means, and sample covariance matrices.

The problems involved in solving these implicit equations are similar to the problems discussed in Section 6.4.1, with the additional complication of having to avoid singular solutions. Of the various techniques that can be used to obtain a solution, the most obvious approach is to use initial estimates to evaluate Eq. (17) for $\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})$ and then to use Eqs. (14)–(16) to update these estimates. If the initial estimates are very good, having perhaps been obtained from a fairly large set of labelled samples, convergence can be quite rapid. However, the results do depend upon the starting point, and the problem of multiple solutions is always present. Furthermore, the repeated computation and inversion of the sample covariance matrices can be quite time consuming.

Considerable simplification can be obtained it it is possible to assume that the covariance matrices are diagonal. This has the added virtue of reducing the number of unknown parameters, which is very important when the number of samples is not large. If this assumption is too strong, it still may be possible to obtain some simplification by assuming that the c covariance matrices are equal, which also eliminates the problem of singular solutions. The derivation of the appropriate maximum likelihood equations for this case is treated in Problems 5 and 6.

6.4.4 A Simple Approximate Procedure

Of the various techniques that can be used to simplify the computation and accelerate convergence, we shall briefly consider one elementary, approximate method. From Eq. (17), it is clear that the probability $\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})$ is large when the squared Mahalanobis distance $(\mathbf{x}_k - \hat{\mu}_i)^t\hat{\Sigma}_i^{-1}(\mathbf{x}_k - \hat{\mu}_i)$ is small. Suppose that we merely compute the squared Euclidean distance $\|\mathbf{x}_k - \hat{\mu}_i\|^2$, find the mean $\hat{\mu}_m$ nearest to $\mathbf{x}_k$, and approximate $\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}})$ as

$$\hat{P}(\omega_i \mid \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \approx \begin{cases} 1 & i = m \\ 0 & \text{otherwise.} \end{cases}$$

Then the iterative application of Eq. (15) leads to the following procedure* for finding $\hat{\mu}_1, \ldots, \hat{\mu}_c$:

```
Procedure: Basic Isodata

       1. Choose some initial values for the means $\hat{\mu}_1, \ldots, \hat{\mu}_c$.
Loop:  2. Classify the $n$ samples by assigning them to the class of the
          closest mean.
       3. Recompute the means as the average of the samples in their
          class.
       4. If any mean changed value, go to Loop; otherwise, stop.
```

This is typical of a class of procedures that are known as *clustering* procedures. Later on we shall place it in the class of iterative optimization procedures, since the means tend to move so as to minimize a squared-error

* Throughout this chapter we shall name and describe various iterative procedures as if they were computer programs. All of these procedures have in fact been programmed, often with much more elaborate provisions for doing such things as breaking ties, avoiding trap states, and allowing more sophisticated terminating conditions. Thus, we occasionally include the word "basic" in their names to emphasize the fact that our interest is limited to explaining essential concepts.

criterion function. At the moment we view it merely as an approximate way to obtain maximum likelihood estimates for the means. The values obtained can be accepted as the answer, or can be used as starting points for the more exact computations.

It is interesting to see how this procedure behaves on the example data in Table 6-1. Figure 6.4 shows the sequence of values for $\hat{\mu}_1$ and $\hat{\mu}_2$ obtained for several different starting points. Since interchanging $\hat{\mu}_1$ and $\hat{\mu}_2$ merely interchanges the labels assigned to the data, the trajectories are symmetric about the line $\hat{\mu}_1 = \hat{\mu}_2$. The trajectories lead either to the point $\hat{\mu}_1 = -2.176$, $\hat{\mu}_2 = 1.684$ or to its image. This is close to the solution found by the maximum likelihood method (viz., $\hat{\mu}_1 = -2.130$ and $\hat{\mu}_2 = 1.668$), and the trajectories show a general resemblance to those shown in Figure 6.3 In general, when the overlap between the component densities is small the maximum likelihood approach and the Isodata procedure can be expected to give similar results.

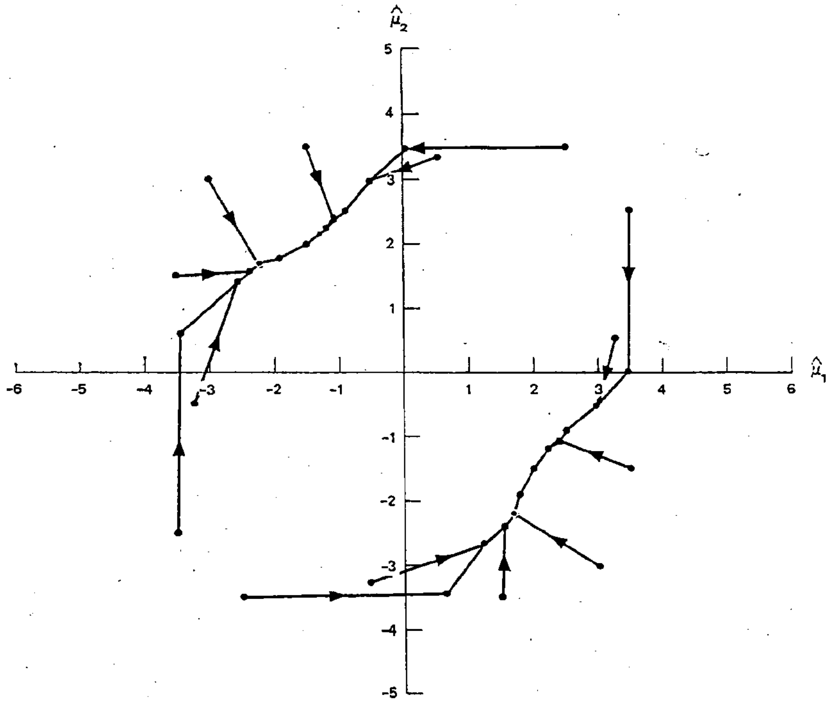


FIGURE 6.4. Trajectories for the Basic Isodata Procedure.

6.5 UNSUPERVISED BAYESIAN LEARNING

6.5.1 The Bayes Classifier

Maximum likelihood methods do not consider the parameter vector $\boldsymbol{\theta}$ to be random—it is just unknown. Prior knowledge about likely values for $\boldsymbol{\theta}$ is irrelevant, although in practice such knowledge may be used in choosing good starting points for hill-climbing procedures. In this section we shall take a Bayesian approach to unsupervised learning. We shall assume that $\boldsymbol{\theta}$ is a random variable with a known a priori distribution $p(\boldsymbol{\theta})$, and we shall use the samples to compute the a posteriori density $p(\boldsymbol{\theta} \mid \mathscr{X})$. Interestingly enough, the analysis will virtually parallel the analysis of supervised Bayesian learning, showing that the two problems are formally very similar.

We begin with an explicit statement of our basic assumptions. We assume that:

1. The number of classes is known.
2. The a priori probabilities $P(\omega_j)$ for each class are known, $j = 1, \ldots, c$.
3. The forms for the class-conditional probability densities $p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j)$ are known, $j = 1, \ldots, c$, but the parameter vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_c)$ is not known.
4. Part of our knowledge about $\boldsymbol{\theta}$ is contained in a known a priori density $p(\boldsymbol{\theta})$.
5. The rest of our knowledge about $\boldsymbol{\theta}$ is contained in a set $\mathscr{X}$ of n samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ drawn independently from the mixture density

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j) P(\omega_j). \tag{1}$$

At this point we could go directly to the calculation of $p(\boldsymbol{\theta} \mid \mathscr{X})$. However, let us first see how this density is used to determine the Bayes classifier. Suppose that a state of nature is selected with probability $P(\omega_i)$ and a feature vector $\mathbf{x}$ is selected according to the probability law $p(\mathbf{x} \mid \omega_i, \boldsymbol{\theta}_i)$. To derive the Bayes classifier we must use all of the information at our disposal to compute the a posteriori probability $P(\omega_i \mid \mathbf{x})$. We exhibit the role of the samples explicitly by writing this as $P(\omega_i \mid \mathbf{x}, \mathscr{X})$. By Bayes rule,

$$P(\omega_i \mid \mathbf{x}, \mathscr{X}) = \frac{p(\mathbf{x} \mid \omega_i, \mathscr{X}) P(\omega_i \mid \mathscr{X})}{\sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \mathscr{X}) P(\omega_j \mid \mathscr{X})}.$$

Since the selection of the state of nature ω_i was done independently of the previously drawn samples, $P(\omega_i \mid \mathscr{X}) = P(\omega_i)$, and we obtain

$$P(\omega_i \mid \mathbf{x}, \mathscr{X}) = \frac{p(\mathbf{x} \mid \omega_i, \mathscr{X})P(\omega_i)}{\sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \mathscr{X})P(\omega_j)}. \tag{18}$$

We introduce the unknown parameter vector by writing

$$\begin{aligned} p(\mathbf{x} \mid \omega_i, \mathscr{X}) &= \int p(\mathbf{x}, \boldsymbol{\theta} \mid \omega_i, \mathscr{X})\, d\boldsymbol{\theta} \\ &= \int p(\mathbf{x} \mid \boldsymbol{\theta}, \omega_i, \mathscr{X}) p(\boldsymbol{\theta} \mid \omega_i, \mathscr{X})\, d\boldsymbol{\theta}. \end{aligned}$$

Since the selection of $\mathbf{x}$ is independent of the samples, $p(\mathbf{x} \mid \boldsymbol{\theta}, \omega_i, \mathscr{X}) = p(\mathbf{x} \mid \omega_i, \boldsymbol{\theta}_i)$. Similarly, since knowledge of the state of nature when $\mathbf{x}$ is selected tells us nothing about the distribution of $\boldsymbol{\theta}$, $p(\boldsymbol{\theta} \mid \omega_i, \mathscr{X}) = p(\boldsymbol{\theta} \mid \mathscr{X})$. Thus we obtain

$$p(\mathbf{x} \mid \omega_i, \mathscr{X}) = \int p(\mathbf{x} \mid \omega_i, \boldsymbol{\theta}_i) p(\boldsymbol{\theta} \mid \mathscr{X})\, d\boldsymbol{\theta}. \tag{19}$$

That is, our best estimate of $p(\mathbf{x} \mid \omega_i)$ is obtained by averaging $p(\mathbf{x} \mid \omega_i, \boldsymbol{\theta}_i)$ over $\boldsymbol{\theta}_i$. Whether or not this is a good estimate depends on the nature of $p(\boldsymbol{\theta} \mid \mathscr{X})$, and thus our attention turns at last to that density.

6.5.2 Learning the Parameter Vector

Using Bayes rule, we can write

$$p(\boldsymbol{\theta} \mid \mathscr{X}) = \frac{p(\mathscr{X} \mid \boldsymbol{\theta})p(\boldsymbol{\theta})}{\int p(\mathscr{X} \mid \boldsymbol{\theta})p(\boldsymbol{\theta})\, d\boldsymbol{\theta}} \tag{20}$$

where the independence of the samples yields

$$p(\mathscr{X} \mid \boldsymbol{\theta}) = \prod_{k=1}^{n} p(\mathbf{x}_k \mid \boldsymbol{\theta}). \tag{21}$$

Alternatively, letting $\mathscr{X}^n$ denote the set of n samples, we can write Eq. (20) in the recursive form

$$p(\boldsymbol{\theta} \mid \mathscr{X}^n) = \frac{p(\mathbf{x}_n \mid \boldsymbol{\theta})p(\boldsymbol{\theta} \mid \mathscr{X}^{n-1})}{\int p(\mathbf{x}_n \mid \boldsymbol{\theta})p(\boldsymbol{\theta} \mid \mathscr{X}^{n-1})\, d\boldsymbol{\theta}}. \tag{22}$$

These are the basic equations for unsupervised Bayesian learning. Eq. (20) emphasizes the relation between the Bayesian and the maximum likelihood solutions. If $p(\boldsymbol{\theta})$ is essentially uniform over the region where $p(\mathscr{X} \mid \boldsymbol{\theta})$ peaks, then $p(\boldsymbol{\theta} \mid \mathscr{X})$ peaks at the same place. If the only significant peak occurs at $\boldsymbol{\theta} = \hat{\boldsymbol{\theta}}$ and if the peak is very sharp, then Eqs. (19) and (18) yield

$$p(\mathbf{x} \mid \omega_i, \mathscr{X}) \approx p(\mathbf{x} \mid \omega_i, \hat{\boldsymbol{\theta}}_i)$$

and

$$P(\omega_i \mid \mathbf{x}, \mathscr{X}) \approx \frac{p(\mathbf{x} \mid \omega_i, \hat{\boldsymbol{\theta}}_i)P(\omega_i)}{\sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \hat{\boldsymbol{\theta}}_j)P(\omega_j)}.$$

That is, these conditions justify the use of the maximum likelihood estimate as if it were the true value of $\boldsymbol{\theta}$ in designing the Bayes classifier.

Of course, if $p(\boldsymbol{\theta})$ has been obtained by supervised learning using a large set of labelled samples, it will be far from uniform, and it will have a dominant influence on $p(\boldsymbol{\theta} \mid \mathscr{X}^n)$, when n is small. Eq. (22) shows how the observation of an additional unlabelled sample modifies our opinion about the true value of $\boldsymbol{\theta}$, and emphasizes the ideas of updating and learning. If the mixture density $p(\mathbf{x} \mid \boldsymbol{\theta})$ is identifiable, then each additional sample tends to sharpen $p(\boldsymbol{\theta} \mid \mathscr{X}^n)$, and under fairly general conditions $p(\boldsymbol{\theta} \mid \mathscr{X}^n)$ can be shown to converge (in probability) to a Dirac delta function centered at the true value of $\boldsymbol{\theta}$. Thus, even though we do not know the categories of the samples, identifiability assures us that we can learn the unknown parameter vector $\boldsymbol{\theta}$, and thereby learn the component densities $p(\mathbf{x} \mid \omega_i, \boldsymbol{\theta})$.

This, then, is the formal Bayesian solution to the problem of unsupervised learning. In retrospect, the fact that unsupervised learning of the parameters of a mixture density is so similar to supervised learning of the parameters of a component density is not at all surprising. Indeed, if the component density is itself a mixture, there would appear to be no essential difference between the two problems.

However, there are some significant differences between supervised and unsupervised learning. One of the major differences concerns the problem of identifiability. With supervised learning, lack of identifiability merely means that instead of obtaining a unique parameter vector we obtain an equivalence class of parameter vectors. However, since all of these yield the same component density, lack of identifiability presents no theoretical difficulty. With unsupervised learning, lack of identifiability is much more serious. When $\boldsymbol{\theta}$ can not be determined uniquely, the mixture can not be decomposed into its true components. Thus, while $p(\mathbf{x} \mid \mathscr{X}^n)$ may still converge to $p(\mathbf{x})$, $p(\mathbf{x} \mid \omega_i, \mathscr{X}^n)$ given by Eq. (19) will not in general converge to $p(\mathbf{x} \mid \omega_i)$, and a theoretical barrier to learning exists.

Another serious problem for unsupervised learning is computational complexity. With supervised learning, the possibility of finding sufficient statistics allows solutions that are analytically pleasing and computationally feasible. With unsupervised learning, there is no way to avoid the fact that the samples are obtained from a mixture density,

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_{j=1}^{c} p(\mathbf{x} \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j), \tag{1}$$

and this gives us little hope of ever finding simple exact solutions for $p(\boldsymbol{\theta} \mid \mathscr{X})$. Such solutions are tied to the existence of a simple sufficient statistic, and the factorization theorem requires the ability to factor $p(\mathscr{X} \mid \boldsymbol{\theta})$ as

$$p(\mathscr{X} \mid \boldsymbol{\theta}) = g(\mathbf{s}, \boldsymbol{\theta})h(\mathscr{X}).$$

But from Eqs. (21) and (1),

$$p(\mathscr{X} \mid \boldsymbol{\theta}) = \prod_{k=1}^{n} \left[\sum_{j=1}^{c} p(\mathbf{x}_k \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j) \right].$$

Thus, $p(\mathscr{X} \mid \boldsymbol{\theta})$ is the sum of c^n products of component densities. Each term in this sum can be interpreted as the joint probability of obtaining the samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ bearing a particular labelling, with the sum extending over all of the ways that the samples could be labelled. Clearly, this results in a thorough mixture of $\boldsymbol{\theta}$ and the $\mathbf{x}$'s, and no simple factoring should be expected. An exception to this statement arises if the component densities do not overlap, so that as $\boldsymbol{\theta}$ varies only one term the mixture density is non-zero. In that case, $p(\mathscr{X} \mid \boldsymbol{\theta})$ is the product of the n nonzero terms, and may possess a simple sufficient statistic. However, since that case allows the class of any sample to be determined, it actually reduces the problem to one of supervised learning, and thus is not a significant exception.

Another way to compare supervised and unsupervised learning is to substitute the mixture density for $p(\mathbf{x}_n \mid \boldsymbol{\theta})$ in Eq. (22) and obtain

$$p(\boldsymbol{\theta} \mid \mathscr{X}^n) = \frac{\sum_{j=1}^{c} p(\mathbf{x}_n \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j)}{\sum_{j=1}^{c} \int p(\mathbf{x}_n \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j)p(\boldsymbol{\theta} \mid \mathscr{X}^{n-1})\, d\boldsymbol{\theta}}\, p(\boldsymbol{\theta} \mid \mathscr{X}^{n-1}). \tag{23}$$

If we consider the special case where $P(\omega_1) = 1$ and all the other a priori probabilities are zero, corresponding to the supervised case in which all samples come from Class 1, then Eq. (23) simplifies to

$$p(\boldsymbol{\theta} \mid \mathscr{X}^n) = \frac{p(\mathbf{x}_n \mid \omega_1, \boldsymbol{\theta}_1)}{\int p(\mathbf{x}_n \mid \omega_1, \boldsymbol{\theta}_1)p(\boldsymbol{\theta} \mid \mathscr{X}^{n-1})\, d\boldsymbol{\theta}}\, p(\boldsymbol{\theta} \mid \mathscr{X}^{n-1}). \tag{24}$$

Let us compare Eqs. (23) and (24) to see how observing an additional sample changes our estimate of $\boldsymbol{\theta}$. In each case we can ignore the denominator, which is independent of $\boldsymbol{\theta}$. Thus, the only significant difference is that in the supervised case we multiply the "a priori" density for $\boldsymbol{\theta}$ by the component density $p(\mathbf{x}_n \mid \omega_1, \boldsymbol{\theta}_1)$, while in the unsupervised case we multiply it by the mixture density $\sum_{j=1}^{c} p(\mathbf{x}_n \mid \omega_j, \boldsymbol{\theta}_j)P(\omega_j)$. Assuming that the sample really did come from Class 1, we see that the effect of not knowing this category membership in the unsupervised case is to diminish the influence of $\mathbf{x}_n$ on changing $\boldsymbol{\theta}$. Since $\mathbf{x}_n$ could have come from any of the c classes, we cannot use it with full effectiveness in changing the component(s) of $\boldsymbol{\theta}$ associated with any one category. Rather, we must distribute its effect over the various categories in accordance with the probability that it arose from each category.

6.5.3 An Example

Consider the one-dimensional, two-component mixture with $p(x \mid \omega_1) \sim N(\mu, 1)$, $p(x \mid \omega_2, \theta) \sim N(\theta, 1)$, where μ, $P(\omega_1)$ and $P(\omega_2)$ are known. Here

$$p(x \mid \theta) = \frac{P(\omega_1)}{\sqrt{2\pi}} \exp[-\tfrac{1}{2}(x - \mu)^2] + \frac{P(\omega_2)}{\sqrt{2\pi}} \exp[-\tfrac{1}{2}(x - \theta)^2].$$

Viewed as a function of x, this mixture density is a superposition of two normal densities, one peaking at $x = \mu$ and the other peaking at $x = \theta$. Viewed as a function of θ, $p(x \mid \theta)$ has a single peak at $\theta = x$. Suppose that the a priori density $p(\theta)$ is uniform from a to b. Then after one observation

$$p(\theta \mid x_1) = \alpha p(x_1 \mid \theta)p(\theta)$$

$$= \begin{cases} \alpha'[P(\omega_1)\exp[-\tfrac{1}{2}(x_1 - \mu)^2] \\ \qquad + P(\omega_2)\exp[-\tfrac{1}{2}(x_1 - \theta)^2]] & a \le \theta \le b \\ 0 & \text{otherwise,} \end{cases}$$

where α and α' are normalizing constants, independent of θ. If the sample x_1 is in the range $a \le x_1 \le b$, then $p(\theta \mid x_1)$ peaks at $\theta = x_1$. Otherwise it peaks either at $\theta = a$ if $x_1 < a$ or at $\theta = b$ if $x_1 > b$. Note that the additive constant $\exp[-(1/2)(x_1 - \mu)^2]$ is large if x_1 is near μ, and thus the peak of $p(\theta \mid x_1)$ is less pronounced if x_1 is near μ. This corresponds to the fact that if x_1 is near μ, it is more likely to have come from the $p(x \mid \omega_1)$ component, and hence its influence on our estimate for θ is diminished.

With the addition of a second sample x_2, $p(\theta \mid x_1)$ changes to

$$p(\theta \mid x_1, x_2) = \beta p(x_2 \mid \theta) p(\theta \mid x_1)$$

$$= \begin{cases} \beta'[P(\omega_1)P(\omega_1)\exp[-\frac{1}{2}(x_1-\mu)^2 - \frac{1}{2}(x_2-\mu)^2] \\ + P(\omega_1)P(\omega_2)\exp[-\frac{1}{2}(x_1-\mu)^2 - \frac{1}{2}(x_2-\theta)^2] \\ + P(\omega_2)P(\omega_1)\exp[-\frac{1}{2}(x_1-\theta)^2 - \frac{1}{2}(x_2-\mu)^2] \\ + P(\omega_2)P(\omega_2)\exp[-\frac{1}{2}(x_1-\theta)^2 - \frac{1}{2}(x_2-\theta)^2]] \\ \hfill a \le \theta \le b \\ 0 \hfill \text{otherwise.} \end{cases}$$

Unfortunately, the primary thing we learn from this expression is that $p(\theta \mid \mathscr{X}^n)$ is already complicated when $n = 2$. The four terms in the sum correspond to the four ways in which the samples could have been drawn from the two component populations. With n samples there will be 2^n terms, and no simple sufficient statistics can be found to facilitate understanding or to simplify computations.

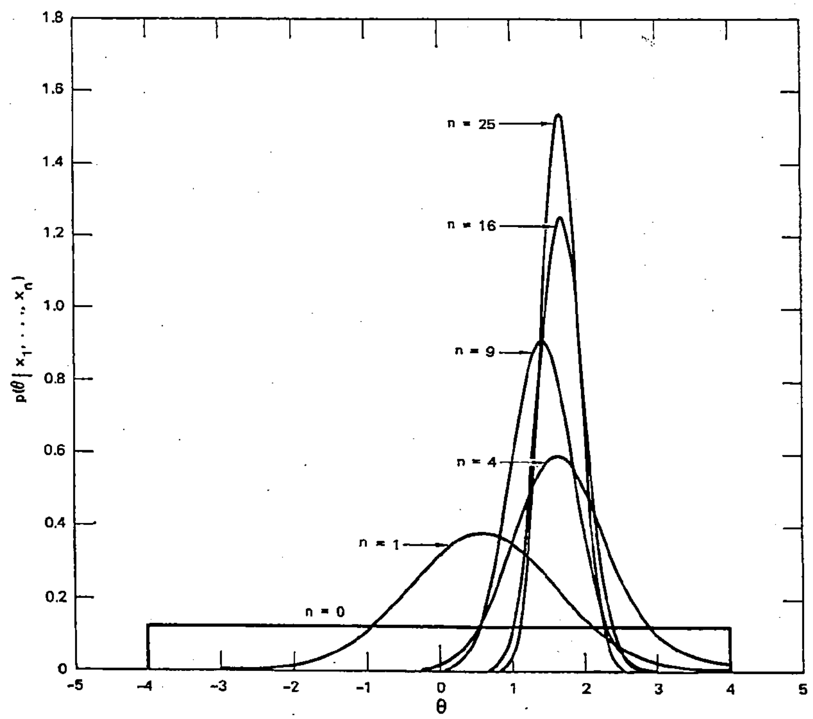


FIGURE 6.5. Unsupervised Bayesian learning.

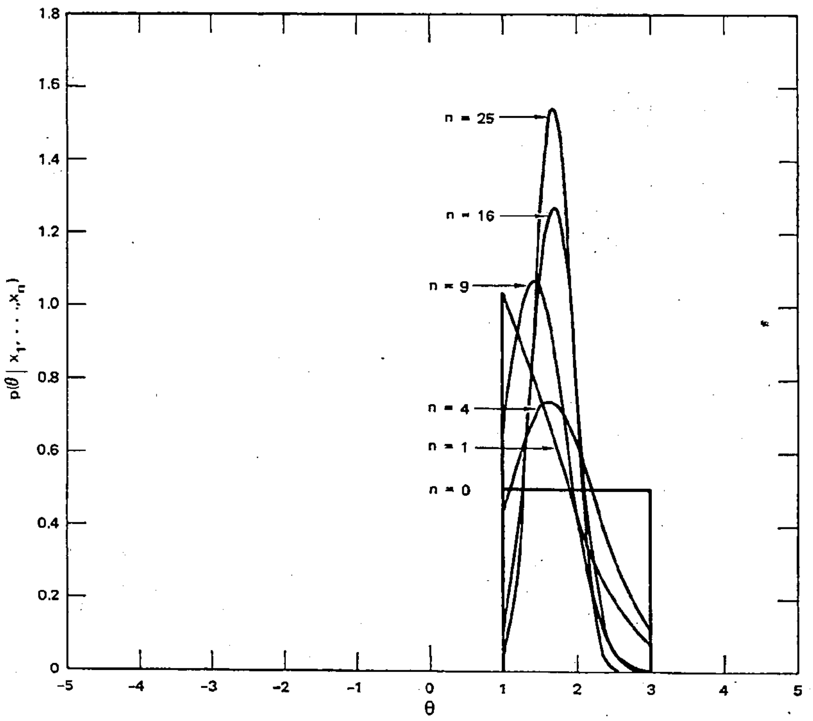


FIGURE 6.6. The effect of narrowing the a priori density.

It is possible to use the relation

$$p(\theta \mid \mathscr{X}^n) = \frac{p(x_n \mid \theta)p(\theta \mid \mathscr{X}^{n-1})}{\int p(x_n \mid \theta)p(\theta \mid \mathscr{X}^{n-1})\, d\theta}$$

and numerical integration to obtain an approximate numerical solution for $p(\theta \mid \mathscr{X}^n)$. This was done for the data in Table 6-1 using the values $\mu = 2$, $P(\omega_1) = 1/3$, and $P(\omega_2) = 2/3$. An a priori density $p(\theta)$ uniform from -4 to 4 encompasses the data in that table. When this was used to start the recursive computation of $p(\theta \mid \mathscr{X}^n)$, the results shown in Figure 6.5 were obtained. As n goes to infinity we can confidently expect $p(\theta \mid \mathscr{X}^n)$ to approach an impulse centered at $\theta = 2$. This graph gives some idea of the rate of convergence.

One of the main differences between the Bayesian and the maximum likelihood approaches to unsupervised learning appears in the presence of the a priori density $p(\theta)$. Figure 6.6 shows how $p(\theta \mid \mathscr{X}^n)$ changes when $p(\theta)$

is assumed to be uniform from 1 to 3, corresponding to more certain initial knowledge about θ. The results of this change are most pronounced when n is small. It is here also that the differences between the Bayesian and the maximum likelihood solutions are most significant. As n increases, the importance of prior knowledge diminishes, and in this particular case the curves for $n = 25$ are virtually identical. In general, one would expect the difference to be small when the number of unlabelled samples is several times the effective number of labelled samples used to determine $p(\theta)$.

6.5.4 Decision-Directed Approximations

Although the problem of unsupervised learning can be stated as merely the problem of estimating parameters of a mixture density, neither the maximum likelihood nor the Bayesian approach yields analytically simple results. Exact solutions for even the simplest nontrivial examples lead to computational requirements that grow exponentially with the number of samples. The problem of unsupervised learning is too important to abandon just because exact solutions are hard to find, however, and numerous procedures for obtaining approximate solutions have been suggested.

Since the basic difference between supervised and unsupervised learning is the presence or absence of labels for the samples, an obvious approach to unsupervised learning is to use the a priori information to design a classifier and to use the decisions of this classifier to label the samples. This is called the *decision-directed* approach to unsupervised learning, and it is subject to many variations. It can be applied sequentially by updating the classifier each time an unlabelled sample is classified. Alternatively, it can be applied in parallel by waiting until all n samples are classified before updating the classifier. If desired, this process can be repeated until no changes occur in the way the samples are labelled.* Various heuristics can be introduced to make the extent of any corrections depend upon the confidence of the classification decision.

There are some obvious dangers associated with the decision-directed approach. If the initial classifier is not reasonably good, or if an unfortunate sequence of samples is encountered, the errors in classifying the unlabelled samples can drive the classifier the wrong way, resulting in a solution corresponding roughly to one of the lesser peaks of the likelihood function. Even if the initial classifier is optimal, the resulting labelling will not in general be the same as the true class membership; the act of classification will exclude samples from the tails of the desired distribution, and will include samples from the tails of the other distributions. Thus, if there is significant

* The Basic Isodata procedure described in Section 6.4.4 is essentially a decision-directed procedure of this type.

overlap between the component densities, one can expect biased estimates and less than optimal results.

Despite these drawbacks, the simplicity of decision-directed procedures makes the Bayesian approach computationally feasible, and a flawed solution is often better than none. If conditions are favorable, performance that is nearly optimal can be achieved at far less computational expense. The literature contains a few rather complicated analyses of particular decision-directed procedures, and numerous reports of experimental results. The basic conclusions are that most of these procedures work well if the parametric assumptions are valid, if there is little overlap between the component densities, and if the initial classifier design is at least roughly correct.

6.6 DATA DESCRIPTION AND CLUSTERING

Let us reconsider our original problem of learning something of use from a set of unlabelled samples. Viewed geometrically, these samples form clouds of points in a d-dimensional space. Suppose that we knew that these points came from a single normal distribution. Then the most we could learn from the data would be contained in the sufficient statistics—the sample mean and the sample covariance matrix. In essence, these statistics constitute a compact description of the data. The sample mean locates the center of gravity of the cloud. It can be thought of as the single point $\mathbf{x}$ that best represents all of the data in the sense of minimizing the sum of squared distances from $\mathbf{x}$ to the samples. The sample covariance matrix tells us how well the sample mean describes the data in terms of the amount of scatter that exists in various directions. If the data points are actually normally distributed, then the cloud has a simple hyperellipsoidal shape, and the sample mean tends to fall in the region where the samples are most densely concentrated.

Of course, if the samples are not normally distributed, these statistics can give a very misleading description of the data. Figure 6.7 shows four different data sets that all have the same mean and covariance matrix. Obviously, second-order statistics are incapable of revealing all of the structure in an arbitrary set of data.

By assuming that the samples come from a mixture of c normal distributions, we can approximate a greater variety of situations. In essence, this corresponds to assuming that the samples fall in hyperellipsoidally-shaped clouds of various sizes and orientations. If the number of component densities is not limited, we can approximate virtually any density function in this way, and use the parameters of the mixture to describe the data. Unfortunately, we have seen that the problem of estimating the parameters of a mixture

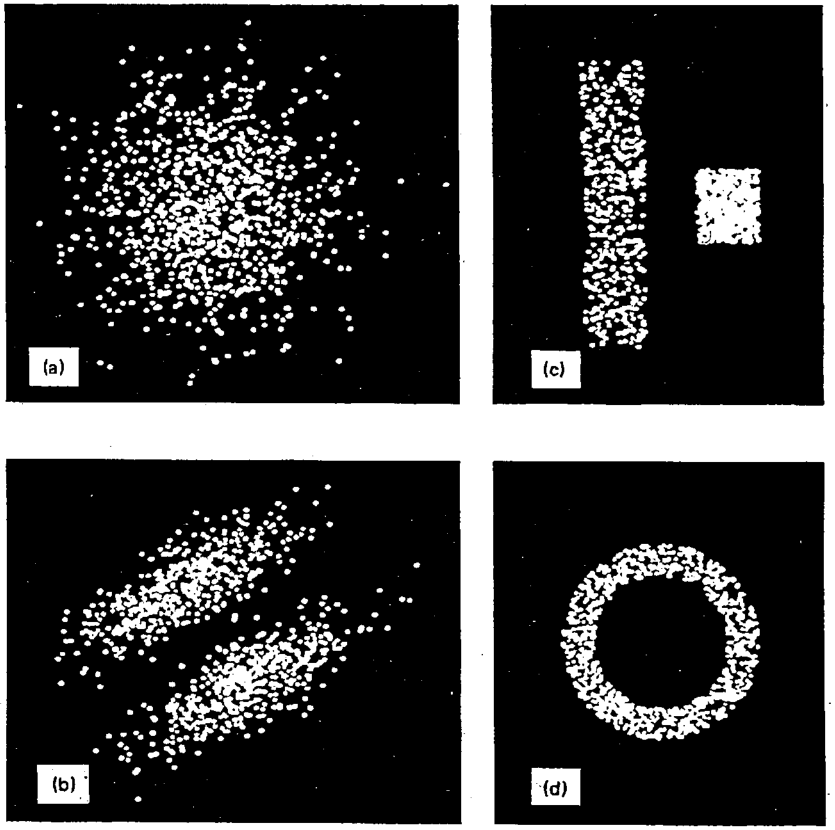


FIGURE 6.7. Data sets having identical second-order statistics.

density is not trivial. Furthermore, in situations where we have relatively little a priori knowledge about the nature of the data, the assumption of particular parametric forms may lead to poor or meaningless results. Instead of finding structure in the data, we would be imposing structure on it.

One alternative is to use one of the nonparametric methods described in Chapter 4 to estimate the unknown mixture density. If accurate, the resulting estimate is certainly a complete description of what we can learn from the data. Regions of high local density, which might correspond to significant subclasses in the population, can be found from the peaks or modes of the estimated density.

If the goal is to find subclasses, a more direct alternative is to use a *clustering procedure*. Roughly speaking, clustering procedures yield a data description in terms of clusters or groups of data points that possess strong internal similarities. The more formal procedures use a criterion function, such as the sum of the squared distances from the cluster centers, and seek the grouping that extremizes the criterion function. Because even this can lead to unmanageable computational problems, other procedures have been proposed that are intuitively appealing but that lead to solutions having no established properties. Their use is usually justified on the ground that they are easy to apply and often yield interesting results that may guide the application of more rigorous procedures.

6.7 SIMILARITY MEASURES

Once we describe the clustering problem as one of finding natural groupings in a set of data, we are obliged to define what we mean by a natural grouping. In what sense are we to say that the samples in one cluster are more like one another than like samples in other clusters? This question actually involves two separate issues—how should one measure the similarity between samples, and how should one evaluate a partitioning of a set of samples into clusters? In this section we address the first of these issues.

The most obvious measure of the similarity (or dissimilarity) between two samples is the distance between them. One way to begin a clustering investigation is to define a suitable distance function and compute the matrix of distances between all pairs of samples. If distance is a good measure of dissimilarity, then one would expect the distance between samples in the same cluster to be significantly less than the distance between samples in different clusters.

Suppose for the moment that we say that two samples belong to the same cluster if the Euclidean distance between them is less than some threshold distance d_0. It is immediately obvious that the choice of d_0 is very important. If d_0 is very large, all of the samples will be assigned to one cluster. If d_0 is very small, each sample will form an isolated cluster. To obtain "natural" clusters, d_0 will have to be greater than typical within-cluster distances and less than typical between-cluster distances (see Figure 6.8).

Less obvious perhaps is the fact that the results of clustering depend on the choice of Euclidean distance as a measure of dissimilarity. This choice implies that the feature space is isotropic. Consequently, clusters defined by Euclidean distance will be invariant to translations or rotations—rigid-body motions of the data points. However, they will not be invariant to linear transformations in general, or to other transformations that distort the

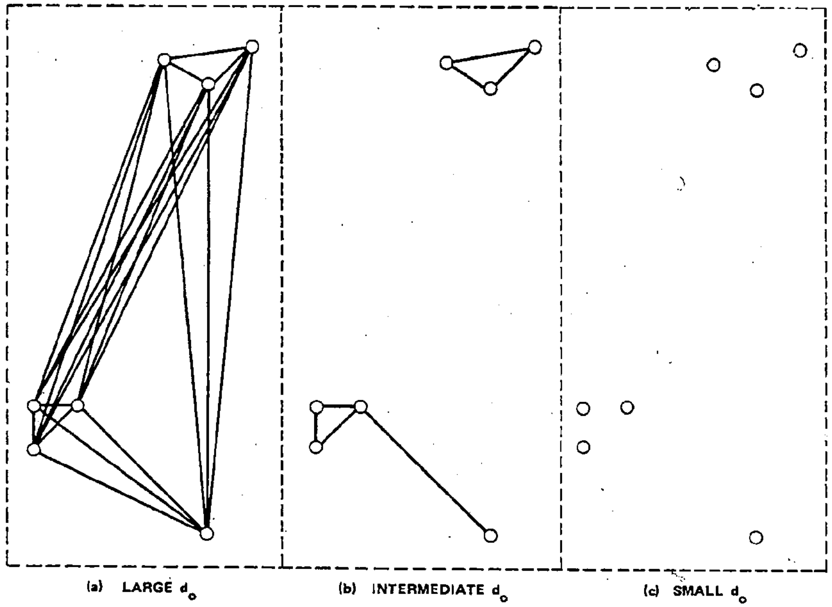


FIGURE 6.8. The effect of a distance threshold on clustering (Lines are drawn between points closer than a distance d_0 apart).

distance relationships. Thus, as Figure 6.9 illustrates, a simple scaling of the coordinate axes can result in a different grouping of the data into clusters. Of course, this is of no concern for problems in which arbitrary rescaling is an unnatural or meaningless transformation. However, if clusters are to mean anything, they should be invariant to transformations natural to the problem.

One way to achieve invariance is to normalize the data prior to clustering. For example, to obtain invariance to displacement and scale changes, one might translate and scale the axes so that all of the features have zero mean and unit variance. To obtain invariance to rotation, one might rotate the axes so that they coincide with the eigenvectors of the sample covariance matrix. This transformation to *principal components* can be preceded and/or followed by normalization for scale.

However, the reader should not conclude that this kind of normalization is necessarily desirable. Consider, for example, the matter of translating and scaling the axes so that each feature has zero mean and unit variance. The rationale usually given for this normalization is that it prevents certain features from dominating distance calculations merely because they have large numerical values. Subtracting the mean and dividing by the standard deviation is an appropriate normalization if this spread of values is due to normal random variation; however, it can be quite inappropriate if the spread is due to the presence of subclasses (see Figure 6.10). Thus, this routine normalization may be less than helpful in the cases of greatest interest. Section 6.8.3 describes some better ways to obtain invariance to scaling.

An alternative to normalizing the data and using Euclidean distance is to use some kind of normalized distance, such as the Mahalanobis distance. More generally, one can abandon the use of distance altogether and introduce a nonmetric *similarity function* $s(\mathbf{x}, \mathbf{x}')$ to compare two vectors $\mathbf{x}$ and $\mathbf{x}'$. Conventionally, this is a symmetric function whose value is large when $\mathbf{x}$ and $\mathbf{x}'$ are similar. For example, when the angle between two vectors is a

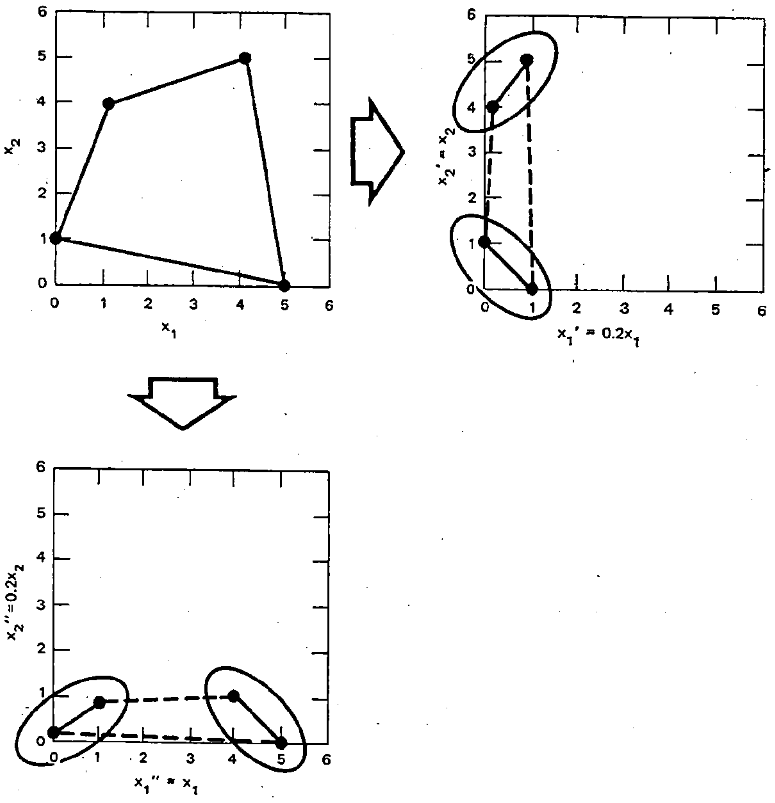
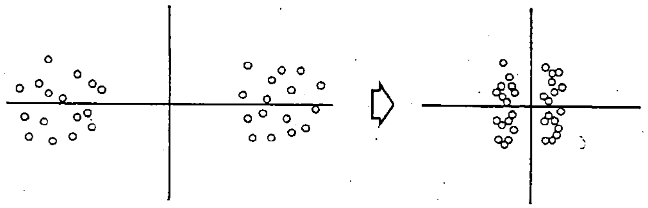


FIGURE 6.9. The effect of scaling on the apparent clustering.



(a) UNNORMALIZED (b) NORMALIZED

FIGURE 6.10. Undesirable effects of normalization.

meaningful measure of their similarity, then the normalized inner product

$$s(\mathbf{x}, \mathbf{x}') = \frac{\mathbf{x}^t\mathbf{x}'}{\|\mathbf{x}\|\,\|\mathbf{x}'\|}$$

may be an appropriate similarity function. This measure, which is the cosine of the angle between $\mathbf{x}$ and $\mathbf{x}'$, is invariant to rotation and dilation, though it is not invariant to translation and general linear transformations.

When the features are binary valued (0 or 1), this similarity function has a simple nongeometrical interpretation in terms of measuring shared features or shared attributes. Let us say that a sample $\mathbf{x}$ *possesses* the *i*th attribute if $x_i = 1$. Then $\mathbf{x}^t\mathbf{x}'$ is merely the number of attributes possessed by $\mathbf{x}$ and $\mathbf{x}'$, and $\|\mathbf{x}\|\,\|\mathbf{x}'\| = (\mathbf{x}^t\mathbf{x}\mathbf{x}'^t\mathbf{x}')^{1/2}$ is the geometric mean of the number of attributes possessed by $\mathbf{x}$ and the number possessed by $\mathbf{x}'$. Thus, $s(\mathbf{x}, \mathbf{x}')$ is a measure of the relative possession of common attributes. Some simple variations are

$$s(\mathbf{x}, \mathbf{x}') = \frac{\mathbf{x}^t\mathbf{x}'}{d},$$

the fraction of attributes shared, and

$$s(\mathbf{x}, \mathbf{x}') = \frac{\mathbf{x}^t\mathbf{x}'}{\mathbf{x}^t\mathbf{x} + \mathbf{x}'^t\mathbf{x}' - \mathbf{x}^t\mathbf{x}'},$$

the ratio of the number of shared attributes to the number possessed by $\mathbf{x}$ or $\mathbf{x}'$. This latter measure (sometimes known as the Tanimoto coefficient) is frequently encountered in the fields of information retrieval and biological taxonomy. Other measures of similarity arise in other applications, the variety of measures testifying to the diversity of problem domains.

We feel obliged to mention that fundamental issues in measurement theory are involved in the use of any distance or similarity function. The calculation of the similarity between two vectors always involves combining the values of their components. Yet, in many pattern recognition applications the components of the feature vector measure seemingly noncomparable quantities. Using our early example of classifying lumber, how can one compare the brightness to the straightness-of-grain? Should the comparison depend on whether the brightness is measured in candles/m^2 or in foot-lamberts? How does one treat vectors whose components have a mixture of nominal, ordinal, interval, and ratio scales?* Ultimately, there is no methodological answer to these questions. When a user selects a particular similarity function or normalizes his data in a particular way, he introduces information that gives the procedure meaning. We have given examples of some alternatives that have proved to be useful. Beyond that we can do little more than alert the unwary to these pitfalls of clustering.

6.8 CRITERION FUNCTIONS FOR CLUSTERING

Suppose that we have a set $\mathscr{X}$ of n samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ that we want to partition into exactly c disjoint subsets $\mathscr{X}_1, \ldots, \mathscr{X}_c$. Each subset is to represent a cluster, with samples in the same cluster being somehow more similar than samples in different clusters. One way to make this into a well-defined problem is to define a criterion function that measures the clustering quality of any partition of the data. Then the problem is one of finding the partition that extremizes the criterion function. In this section we examine the characteristics of several basically similar criterion functions, postponing until later the question of how to find an optimal partition.

6.8.1 The Sum-of-Squared-Error Criterion

The simplest and most widely used criterion function for clustering is the sum-of-squared-error criterion. Let n_i be the number of samples in $\mathscr{X}_i$ and let $\mathbf{m}_i$ be the mean of those samples,

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{\mathbf{x} \in \mathscr{X}_i} \mathbf{x}. \tag{25}$$

Then the sum of squared errors is defined by

$$J_e = \sum_{i=1}^{c} \sum_{\mathbf{x} \in \mathscr{X}_i} \|\mathbf{x} - \mathbf{m}_i\|^2. \tag{26}$$

This criterion function has a simple interpretation. For a given cluster $\mathscr{X}_i$, the mean vector $\mathbf{m}_i$ is the best representative of the samples in $\mathscr{X}_i$ in the sense that it minimizes the sum of the squared lengths of the "error" vectors $\mathbf{x} - \mathbf{m}_i$. Thus, J_e measures the total squared error incurred in representing the n samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ by the c cluster centers $\mathbf{m}_1, \ldots, \mathbf{m}_c$. The value of

* These fundamental considerations are by no means unique to clustering. They appear, for example, whenever one chooses a parametric form for an unknown probability density function, a metric for nonparametric density estimation, or scale factors for linear discriminant functions. Clustering problems merely expose them more clearly.

J_e depends on how the samples are grouped into clusters, and an optimal partitioning is defined as one that minimizes J_e. Clusterings of this type are often called *minimum variance* partitions.

What kind of clustering problems are well suited to a sum-of-squared-error criterion? Basically, J_e is an appropriate criterion when the clusters form essentially compact clouds that are rather well separated from one another. It should work well for the two or three clusters in Figure 6.11, but one would not expect reasonable results for the data in Figure 6.12.* A less obvious problem arises when there are great differences in the number of samples in different clusters. In that case it can happen that a partition that splits a large cluster is favored over one that maintains the integrity of the clusters merely because the slight reduction in squared error achieved is multiplied by many terms in the sum (see Figure 6.13). This situation frequently arises because of the presence of "outliers" or "wild shots," and brings up the problem of interpreting and evaluating the results of clustering. Since little can be said about that problem, we shall merely observe that if additional considerations render the results of minimizing J_e unsatisfactory, then these considerations should be used, if possible, in formulating a better criterion function.

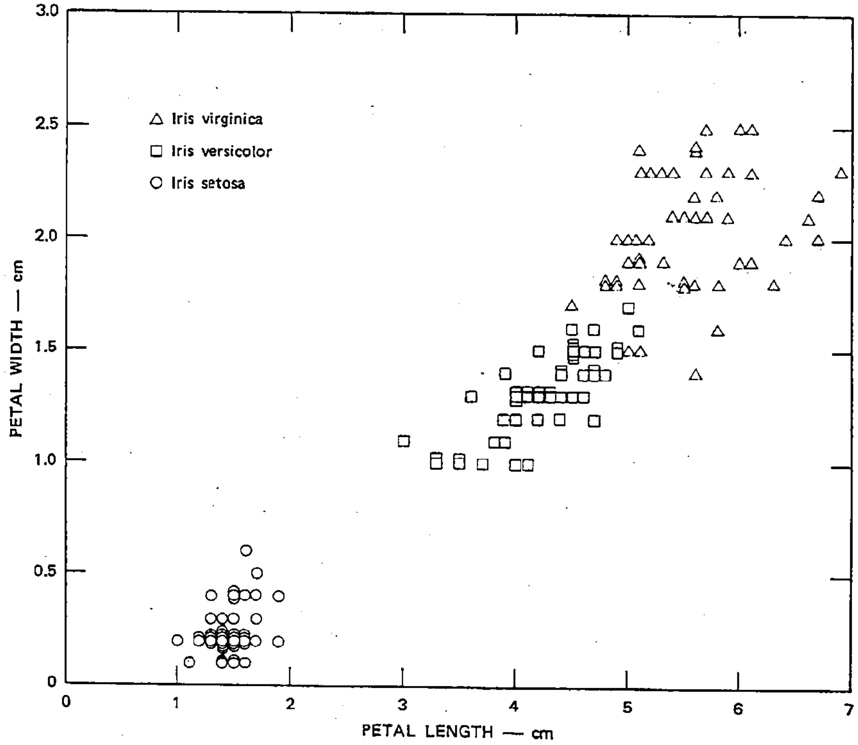


FIGURE 6.11. A two-dimensional section of the Anderson iris data.

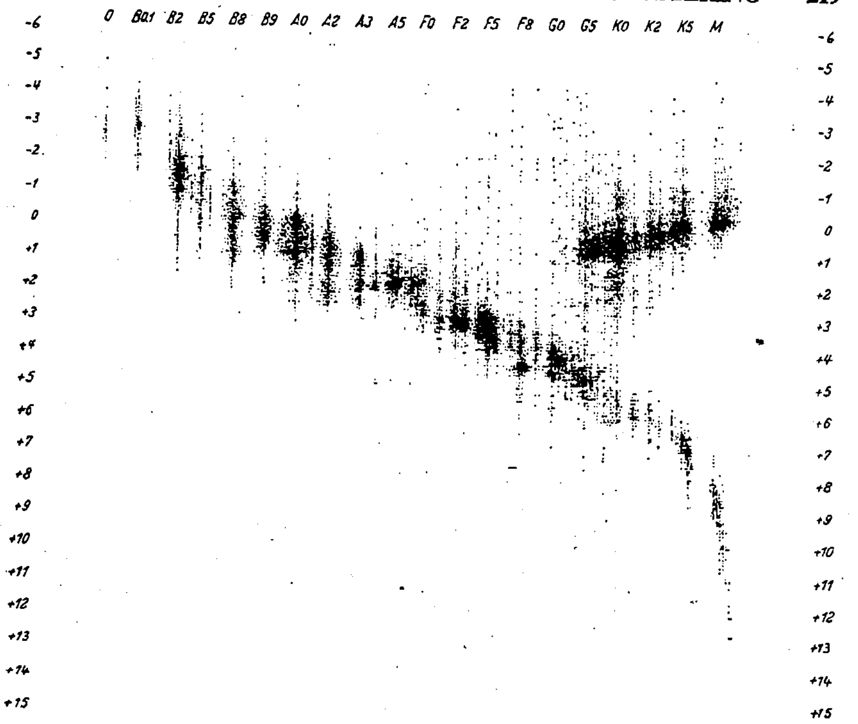


FIGURE 6.12. The Herzsprung-Russell Diagram (Courtesy Lunds Universitet Institutionen für Astronomi).

6.8.2 Related Minimum Variance Criteria

By some simple algebraic manipulation we can eliminate the mean vectors from the expression for J_e and obtain the equivalent expression

$$J_e = \frac{1}{2} \sum_{i=1}^{c} n_i \bar{s}_i, \tag{27}$$

* These two data sets are well known for quite different reasons. Figure 6.11 shows two of four measurements made by E. Anderson on 150 samples of three species of iris. These data were listed and used by R. A. Fisher in his classic paper on discriminant analysis (Fisher 1936), and have since become a favorite example for illustrating clustering procedures. Figure 6.12 is well known in astronomy as the Hertzsprung and Russell (or spectrum-luminosity) diagram, which led to the subdivision of stars into such categories as giants, supergiants, main sequence stars, and dwarfs. It was used by E. W. Forgey and again by D. Wishart (1969) to illustrate the limitations of simple clustering procedures.

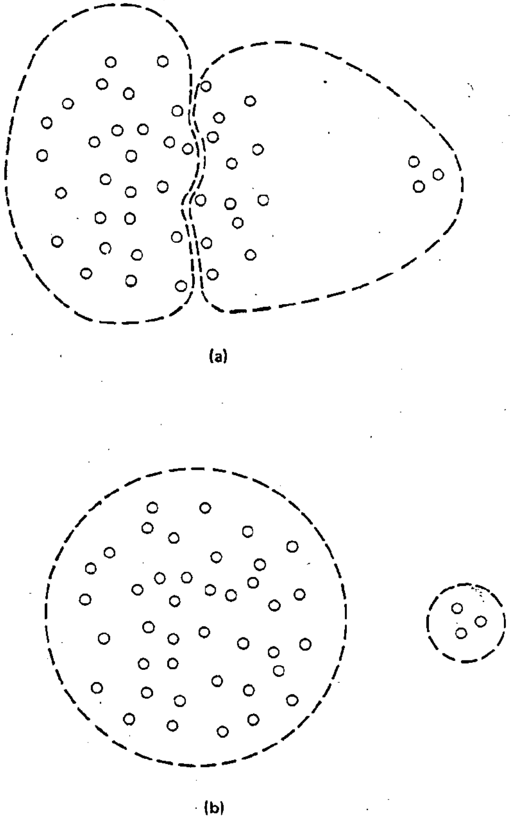


FIGURE 6.13. The problem of splitting large clusters: the sum of squared error is smaller for (a) than for (b).

where

$$\bar{s}_i = \frac{1}{n_i^2} \sum_{\mathbf{x}\in\mathscr{X}_i} \sum_{\mathbf{x}'\in\mathscr{X}_i} \|\mathbf{x} - \mathbf{x}'\|^2. \tag{28}$$

Eq. (28) leads us to interpret $\bar{s}_i$ as the average squared distance between points in the ith cluster, and emphasizes the fact that the sum-of-squared-error criterion uses Euclidean distance as the measure of similarity. It also suggests an obvious way of obtaining other criterion functions. For example, one can replace $\bar{s}_i$ by the average, the median, or perhaps the maximum distance between points in a cluster. More generally, one can introduce an appropriate

similarity function $s(\mathbf{x}, \mathbf{x}')$ and replace $\bar{s}_i$ by functions such as

$$\bar{s}_i = \frac{1}{n_i^2} \sum_{\mathbf{x}\in\mathscr{X}_i} \sum_{\mathbf{x}'\in\mathscr{X}_i} s(\mathbf{x}, \mathbf{x}') \tag{29}$$

or

$$\bar{s}_i = \min_{\mathbf{x},\mathbf{x}'\in\mathscr{X}_i} s(\mathbf{x}, \mathbf{x}'). \tag{30}$$

As before, we define an optimal partitioning as one that extremizes the criterion function. This creates a well-defined problem, and the hope is that its solution discloses the intrinsic structure of the data.

6.8.3 Scattering Criteria

6.8.3.1 THE SCATTER MATRICES

Another interesting class of criterion functions can be derived from the scatter matrices used in multiple discriminant analysis. The following definitions directly parallel the definitions given in Section 4.11.

Mean vector for ith cluster:

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{\mathbf{x}\in\mathscr{X}_i} \mathbf{x}. \tag{31}$$

Total mean vector:

$$\mathbf{m} = \frac{1}{n} \sum_{\mathscr{X}} \mathbf{x} = \frac{1}{n} \sum_{i=1}^{c} n_i \mathbf{m}_i. \tag{32}$$

Scatter matrix for ith cluster:

$$S_i = \sum_{\mathbf{x}\in\mathscr{X}_i} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^t. \tag{33}$$

Within-cluster scatter matrix:

$$S_W = \sum_{i=1}^{c} S_i. \tag{34}$$

Between-cluster scatter matrix:

$$S_B = \sum_{i=1}^{c} n_i(\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^t. \tag{35}$$

Total scatter matrix:

$$S_T = \sum_{\mathbf{x}\in\mathscr{X}} (\mathbf{x} - \mathbf{m})(\mathbf{x} - \mathbf{m})^t. \tag{36}$$

As before, it follows from these definitions that the total scatter matrix is the sum of the within-cluster scatter matrix and the between-cluster scatter matrix:

$$S_T = S_W + S_B. \tag{37}$$

Note that the total scatter matrix does not depend on how the set of samples is partitioned into clusters. It depends only on the total set of samples. The within-cluster and between-cluster scatter matrices do depend on the partitioning, however. Roughly speaking, there is an exchange between these two matrices, the between-cluster scatter going up as the within-cluster scatter goes down. This is fortunate, since by trying to minimize the within-cluster scatter we will also tend to maximize the between-cluster scatter.

To be more precise in talking about the amount of within-cluster or between-cluster scatter, we need a scalar measure of the "size" of a scatter matrix. The two measures that we shall consider are the *trace* and the *determinant*. In the univariate case, these two measures are equivalent, and we can define an optimal partition as one that minimizes S_W or maximizes S_B. In the multivariate case things are somewhat more complicated, and a number of related but distinct optimality criteria have been suggested.

6.8.3.2 THE TRACE CRITERION

Perhaps the simplest scalar measure of a scatter matrix is its trace, the sum of its diagonal elements. Roughly speaking, the trace measures the square of the scattering radius, since it is proportional to the sum of the variances in the coordinate directions. Thus, an obvious criterion function to minimize is the trace of S_W. In fact, this criterion is nothing more or less than the sum-of-squared-error criterion, since Eqs. (33) and (34) yield

$$\operatorname{tr} S_W = \sum_{i=1}^{c} \operatorname{tr} S_i = \sum_{i=1}^{c} \sum_{\mathbf{x}\in\mathscr{X}_i} \|\mathbf{x} - \mathbf{m}_i\|^2 = J_e. \tag{38}$$

Since $\operatorname{tr} S_T = \operatorname{tr} S_W + \operatorname{tr} S_B$ and $\operatorname{tr} S_T$ is independent of how the samples are partitioned, we see that no new results are obtained by trying to maximize $\operatorname{tr} S_B$. However, it is comforting to know that in trying to minimize the within-cluster criterion $J_e = \operatorname{tr} S_W$ we are also maximizing the between-cluster criterion

$$\operatorname{tr} S_B = \sum_{i=1}^{c} n_i \|\mathbf{m}_i - \mathbf{m}\|^2. \tag{39}$$

6.8.3.3 THE DETERMINANT CRITERION

In Section 4.11 we used the determinant of the scatter matrix to obtain a scalar measure of scatter. Roughly speaking, this measures the square of the scattering volume, since it is proportional to the product of the variances in the directions of the principal axes. Since S_B will be singular if the number of clusters is less than or equal to the dimensionality, $|S_B|$ is obviously a poor choice for a criterion function. S_W can also become singular, and will

certainly be so if $n - c$ is less than the dimensionality d.* However, if we assume that S_W is nonsingular, we are led to consider the criterion function

$$J_d = |S_W| = \left| \sum_{i=1}^{c} S_i \right|. \tag{40}$$

The partition that minimizes J_d is often similar to the one that minimizes J_e, but the two need not be the same. We observed before that the minimum-squared-error partition might change if the axes are scaled. This does not happen with J_d. To see why, let T be a nonsingular matrix and consider the change of variables $\mathbf{x}' = T\mathbf{x}$. Keeping the partitioning fixed, we obtain new mean vectors $\mathbf{m}_i' = T\mathbf{m}_i$ and new scatter matrices $S_i' = TS_iT^t$. Thus, J_d changes to

$$J_d' = |S_W'| = |TS_WT^t| = |T|^2 J_d.$$

Since the scale factor $|T|^2$ is the same for all partitions, it follows that J_d and J_d' rank the partitions in the same way, and hence that the optimal clustering based on J_d is invariant to nonsingular linear transformations of the data.

6.8.3.4 INVARIANT CRITERIA

It is not hard to show that the eigenvalues $\lambda_1, \ldots, \lambda_d$ of $S_W^{-1}S_B$ are invariant under nonsingular linear transformations of the data. Indeed, these eigenvalues are the basic linear invariants of the scatter matrices. Their numerical values measure the ratio of between-cluster to within-cluster scatter in the direction of the eigenvectors, and partitions that yield large values are usually desirable. Of course, as we pointed out in Section 4.11, the fact that the rank of S_B can not exceed $c - 1$ means that no more than $c - 1$ of these eigenvalues can be nonzero. Nevertheless, good partitions are ones for which the nonzero eigenvalues are large.

One can invent a great variety of invariant clustering criteria by composing appropriate functions of these eigenvalues. Some of these follow naturally from standard matrix operations. For example, since the trace of a matrix is the sum of its eigenvalues, one might elect to maximize the criterion function†

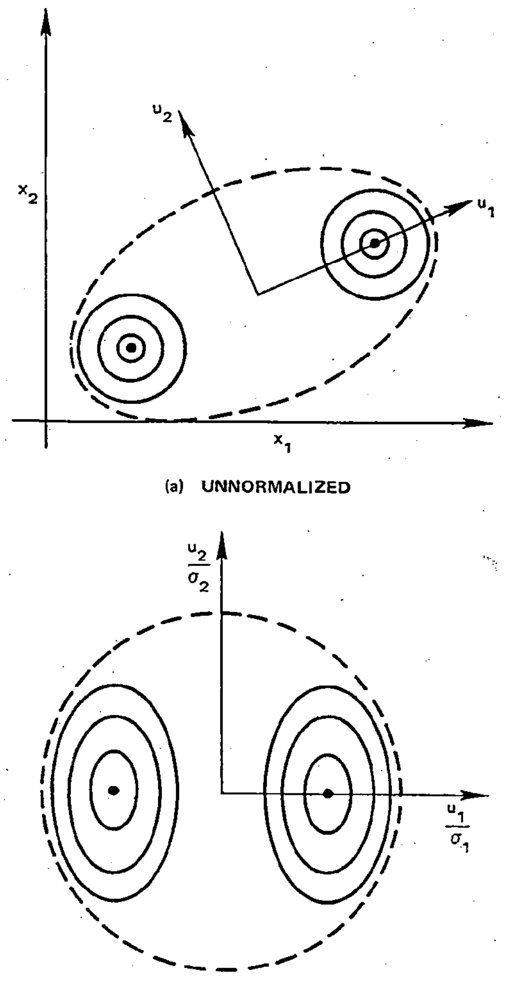
$$\operatorname{tr} S_W^{-1}S_B = \sum_{i=1}^{d} \lambda_i. \tag{41}$$

* This follows from the fact that the rank of S_i can not exceed $n_i - 1$, and thus the rank of S_W can not exceed $\Sigma(n_i - 1) = n - c$. Of course, if the samples are confined to a lower dimensional subspace it is possible to have S_W be singular even though $n - c \geq d$. In such cases, some kind of dimensionality-reduction procedure must be used before the determinant criterion can be applied (see Section 6.14).

† Another invariant criterion is

$$|S_W^{-1}S_B| = \prod_{i=1}^{d} \lambda_i.$$

However, since its value is usually zero it is not very useful.



(a) UNNORMALIZED

(b) NORMALIZED

FIGURE 6.14. The effect of transforming to normalized principal components (Note: the partition that minimizes $S_T^{-1}S_W$ in (a) minimizes the sum of squared errors in (b).).

By using the relation $S_T = S_W + S_B$, one can derive the following invariant relatives of tr S_W and $|S_W|$:

$$\operatorname{tr} S_T^{-1}S_W = \sum_{i=1}^{d} \frac{1}{1+\lambda_i} \tag{42}$$

$$\frac{|S_W|}{|S_T|} = \prod_{i=1}^{d} \frac{1}{1+\lambda_i}. \tag{43}$$

Since all of these criterion functions are invariant to linear transformations, the same is true of the partitions that extremize them. In the special case of two clusters, only one eigenvalue is nonzero, and all of these criteria yield the same clustering. However, when the samples are partitioned into more than two clusters, the optimal partitions, though often similar, need not be the same.

With regard to the criterion functions involving S_T, note that S_T does not depend on how the samples are partitioned into clusters. Thus, the clusterings that minimize $|S_W|/|S_T|$ are exactly the same as the ones that minimize $|S_W|$. If we rotate and scale the axes so that S_T becomes the identity matrix, we see that minimizing tr $S_T^{-1}S_W$ is equivalent to minimizing the sum-of-squared-error criterion tr S_W after performing this normalization. Figure 6.14 illustrates the effects of this transformation graphically. Clearly, this criterion suffers from the very defects that we warned about in Section 6.7, and it is probably the least desirable of these criteria.

One final warning about invariant criteria is in order. If different apparent groupings can be obtained by scaling the axes or by applying any other linear transformation, then all of these groupings will be exposed by invariant procedures. Thus, invariant criterion functions are more likely to possess multiple local extrema, and are correspondingly more difficult to extremize.

The variety of the criterion functions we have discussed and the somewhat subtle differences between them should not be allowed to obscure their essential similarity. In every case the underlying model is that the samples form c fairly well separated clouds of points. The within-cluster scatter matrix S_W is used to measure the compactness of these clouds, and the basic goal is to find the most compact grouping. While this approach has proved useful for many problems, it is not universally applicable. For example, it will not extract a very dense cluster embedded in the center of a diffuse cluster, or separate intertwined line-like clusters. For such cases one must devise other criterion functions that are better matched to the structure present or being sought.

6.9 ITERATIVE OPTIMIZATION

Once a criterion function has been selected, clustering becomes a well-defined problem in discrete optimization: find those partitions of the set of samples that extremize the criterion function. Since the sample set is finite, there are only a finite number of possible partitions. Thus, in theory the clustering problem can always be solved by exhaustive enumeration. However, in practice such an approach is unthinkable for all but the simplest problems. There are approximately $c^n/c!$ ways of partitioning a set of n elements into c subsets,† and this exponential growth with n is overwhelming. For example, an exhaustive search for the best set of 5 clusters in 100 samples would require considering more than 10^{67} partitionings. Thus, in most applications an exhaustive search is completely infeasible.

The approach most frequently used in seeking optimal partitions is iterative optimization. The basic idea is to find some reasonable initial partition and to "move" samples from one group to another if such a move will improve the value of the criterion function. Like hill-climbing procedures in general, these approaches guarantee local but not global optimization. Different starting points can lead to different solutions, and one never knows whether or not the best solution has been found. Despite these limitations, the fact that the computational requirements are bearable makes this approach significant.

Let us consider the use of iterative improvement to minimize the sum-of-squared-error criterion J_e, written as

$$J_e = \sum_{i=1}^{c} J_i,$$

where

$$J_i = \sum_{\mathbf{x} \in \mathscr{X}_i} \|\mathbf{x} - \mathbf{m}_i\|^2$$

and

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{\mathbf{x} \in \mathscr{X}_i} \mathbf{x}.$$

Suppose that a sample $\hat{\mathbf{x}}$ currently in cluster $\mathscr{X}_i$ is tentatively moved to $\mathscr{X}_j$. Then $\mathbf{m}_j$ changes to

$$\mathbf{m}_j^* = \mathbf{m}_j + \frac{\hat{\mathbf{x}} - \mathbf{m}_j}{n_j + 1}$$

† The reader who likes combinatorial problems will enjoy showing that there are exactly

$$\frac{1}{c!} \sum_{i=1}^{c} \binom{c}{i} (-1)^{c-i} i^n$$

partitions of n items into c nonempty subsets. (see W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. I, p. 58 (John Wiley, New York, Second Edition, 1959)). If $n \gg c$, the last term is the most significant.

and J_j increases to

$$\begin{aligned} J_j^* &= \sum_{\mathbf{x} \in \mathscr{X}_j} \|\mathbf{x} - \mathbf{m}_j^*\|^2 + \|\hat{\mathbf{x}} - \mathbf{m}_j^*\|^2 \\ &= \sum_{\mathbf{x} \in \mathscr{X}_j} \left\| \mathbf{x} - \mathbf{m}_j - \frac{\hat{\mathbf{x}} - \mathbf{m}_j}{n_j + 1} \right\|^2 + \left\| \frac{n_j}{n_j + 1} (\hat{\mathbf{x}} - \mathbf{m}_j) \right\|^2 \\ &= J_j + \frac{n_j}{n_j + 1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2. \end{aligned}$$

Under the assumption that $n_i \neq 1$ (singleton clusters should not be destroyed), a similar calculation shows that $\mathbf{m}_i$ changes to

$$\mathbf{m}_i^* = \mathbf{m}_i - \frac{\hat{\mathbf{x}} - \mathbf{m}_i}{n_i - 1}$$

and J_i decreases to

$$J_i^* = J_i - \frac{n_i}{n_i - 1} \|\hat{\mathbf{x}} - \mathbf{m}_i\|^2.$$

These equations greatly simplify the computation of the change in the criterion function. The transfer of $\hat{\mathbf{x}}$ from $\mathscr{X}_i$ to $\mathscr{X}_j$ is advantageous if the decrease in J_i is greater than the increase in J_j. This is the case if

$$n_i/(n_i - 1) \|\hat{\mathbf{x}} - \mathbf{m}_i\|^2 > n_j/(n_j + 1) \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2,$$

which typically happens whenever $\hat{\mathbf{x}}$ is closer to $\mathbf{m}_j$ than $\mathbf{m}_i$. If reassignment is profitable, the greatest decrease in sum of squared error is obtained by selecting the cluster for which $n_j/(n_j + 1) \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2$ is minimum. This leads to the following clustering procedure:

```
Procedure: Basic Minimum Squared Error
           1. Select an initial partition of the $n$ samples into clusters and
              compute $J_e$ and the means $\mathbf{m}_1, \ldots, \mathbf{m}_c$.
Loop:      2. Select the next candidate sample $\hat{\mathbf{x}}$. Suppose that $\hat{\mathbf{x}}$ is currently
              in $\mathscr{X}_i$.
           3. If $n_i = 1$ go to Next; otherwise compute
```

$$\rho_j = \begin{cases} \dfrac{n_j}{n_j + 1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2 & j \neq i \\[2ex] \dfrac{n_i}{n_i - 1} \|\hat{\mathbf{x}} - \mathbf{m}_i\|^2 & j = i. \end{cases}$$

```
           4. Transfer $\hat{\mathbf{x}}$ to $\mathscr{X}_k$ if $\rho_k \leq \rho_j$ for all $j$.
           5. Update $J_e$, $\mathbf{m}_i$, and $\mathbf{m}_k$.
Next:      6. If $J_e$ has not changed in $n$ attempts, stop; otherwise go to Loop.
```

If this procedure is compared to the Basic Isodata procedure described in Section 6.4.4, it is clear that the former is essentially a sequential version of the latter. Where the Basic Isodata procedure waits until all n samples have been reclassified before updating, the Basic Minimum Squared Error procedure updates after each sample is reclassified. It has been experimentally observed that this procedure is more susceptible to being trapped at a local minimum, and it has the further disadvantage of making the results depend on the order in which the candidates are selected. However, it is at least a stepwise optimal procedure, and it can be easily modified to apply to problems in which samples are acquired sequentially and clustering must be done in real time.

One question that plagues all hill-climbing procedures is the choice of the starting point. Unfortunately, there is no simple, universally good solution to this problem. One approach is to select c samples randomly for the initial cluster centers, using them to partition the data on a minimum-distance basis. Repetition with different random selections can give some indication of the sensitivity of the solution to the starting point. Another approach is to find the c-cluster starting point from the solution to the $(c - 1)$-cluster problem. The solution for the one-cluster problem is the total-sample mean; the starting point for the c-cluster problem can be the final means for the $(c - 1)$-cluster problem plus the sample that is furthest from the nearest cluster center. This approach leads us directly to the so-called hierarchical clustering procedures, which are simple methods that can provide very good starting points for iterative optimization.

6.10 HIERARCHICAL CLUSTERING

6.10.1 Definitions

Let us consider a sequence of partitions of the n samples into c clusters. The first of these is a partition into n clusters, each cluster containing exactly one sample. The next is a partition into $n - 1$ clusters, the next a partition into $n - 2$, and so on until the nth, in which all the samples form one cluster. We shall say that we are at level k in the sequence when $c = n - k + 1$. Thus, level one corresponds to n clusters and level n to one. Given any two samples $\mathbf{x}$ and $\mathbf{x}'$, at some level they will be grouped together in the same cluster. If the sequence has the property that whenever two samples are in the same cluster at level k they remain together at all higher levels, then the sequence is said to be a *hierarchical clustering*. The classical examples of hierarchical clustering appear in biological taxonomy, where individuals are grouped into species, species into genera, genera into families, and so on.

In fact, this kind of clustering permeates classificatory activities in the sciences.

For every hierarchical clustering there is a corresponding tree, called a *dĕndrogram*, that shows how the samples are grouped. Figure 6.15 shows a dendrogram for a hypothetical problem involving six samples. Level 1 shows the six samples as singleton clusters. At level 2, samples x_3 and x_5 have been grouped to form a cluster, and they stay together at all subsequent levels. If it is possible to measure the similarity between clusters, then the dendrogram is usually drawn to scale to show the similarity between the clusters that are grouped. In Figure 6.15, for example, the similarity between the two groups of samples that are merged at level 6 has a value of 30. The similarity-values are often used to help determine whether the groupings are natural or forced. For our hypothetical example, one would be inclined to say that the groupings at levels 4 or 5 are natural, but that the large reduction in similarity needed to go to level 6 makes that grouping forced. We shall see shortly how such similarity values can be obtained.

Because of their conceptual simplicity, hierarchical clustering procedures are among the best-known methods. The procedures themselves can be divided into two distinct classes, agglomerative and divisive. *Agglomerative* (bottom-up, clumping) procedures start with n singleton clusters and form

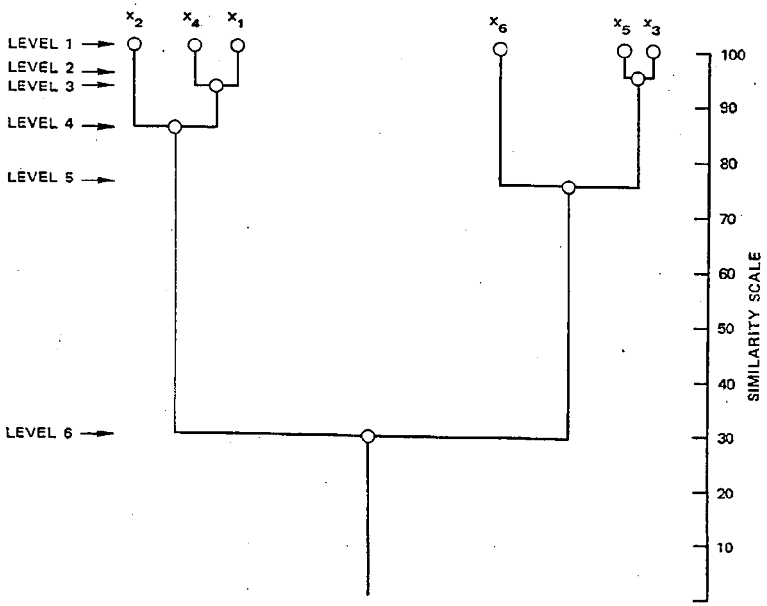


FIGURE 6.15. A dendrogram for hierarchical clustering.

the sequence by successively merging clusters. *Divisive* (top-down, splitting) procedures start with all of the samples in one cluster and form the sequence by successively splitting clusters. The computation needed to go from one level to another is usually simpler for the agglomerative procedures. However, when there are many samples and one is interested in only a small number of clusters, this computation will have to be repeated many times. For simplicity, we shall limit our attention to the agglomerative procedures, referring the reader to the literature for divisive methods.

6.10.2 Agglomerative Hierarchical Clustering

The major steps in agglomerative clustering are contained in the following procedure:

```
Procedure:  Basic Agglomerative Clustering
        1. Let $\hat{c} = n$ and $\mathscr{X}_i = \{\mathbf{x}_i\}$, $i = 1, \ldots, n$.
Loop:   2. If $\hat{c} \leq c$, stop.
        3. Find the nearest pair of distinct clusters, say $\mathscr{X}_i$ and $\mathscr{X}_j$.
        4. Merge $\mathscr{X}_i$ and $\mathscr{X}_j$, delete $\mathscr{X}_j$, and decrement $\hat{c}$ by one.
        5. Go to Loop.
```

As described, this procedure terminates when the specified number of clusters has been obtained. However, if we continue until $c = 1$ we can produce a dendrogram like that shown in Figure 6.15. At any level the distance between nearest clusters can provide the dissimilarity value for that level. The reader will note that we have not said how to measure the distance between two clusters. The considerations here are much like those involved in selecting a criterion function. For simplicity, we shall restrict our attention to the following distance measures, leaving extensions to other similarity measures to the reader's imagination:

$$d_{\min}(\mathscr{X}_i, \mathscr{X}_j) = \min_{\mathbf{x}\in\mathscr{X}_i, \mathbf{x}'\in\mathscr{X}_j} \|\mathbf{x} - \mathbf{x}'\|$$

$$d_{\max}(\mathscr{X}_i, \mathscr{X}_j) = \max_{\mathbf{x}\in\mathscr{X}_i, \mathbf{x}'\in\mathscr{X}_j} \|\mathbf{x} - \mathbf{x}'\|$$

$$d_{\text{avg}}(\mathscr{X}_i, \mathscr{X}_j) = \frac{1}{n_i n_j} \sum_{\mathbf{x}\in\mathscr{X}_i} \sum_{\mathbf{x}'\in\mathscr{X}_j} \|\mathbf{x} - \mathbf{x}'\|$$

$$d_{\text{mean}}(\mathscr{X}_i, \mathscr{X}_j) = \|\mathbf{m}_i - \mathbf{m}_j\|.$$

All of these measures have a minimum-variance flavor, and they usually yield the same results if the clusters are compact and well separated. However, if the clusters are close to one another, or if their shapes are not basically hyperspherical, quite different results can be obtained. We shall use the two-dimensional point sets shown in Figure 6.16 to illustrate some of the differences.

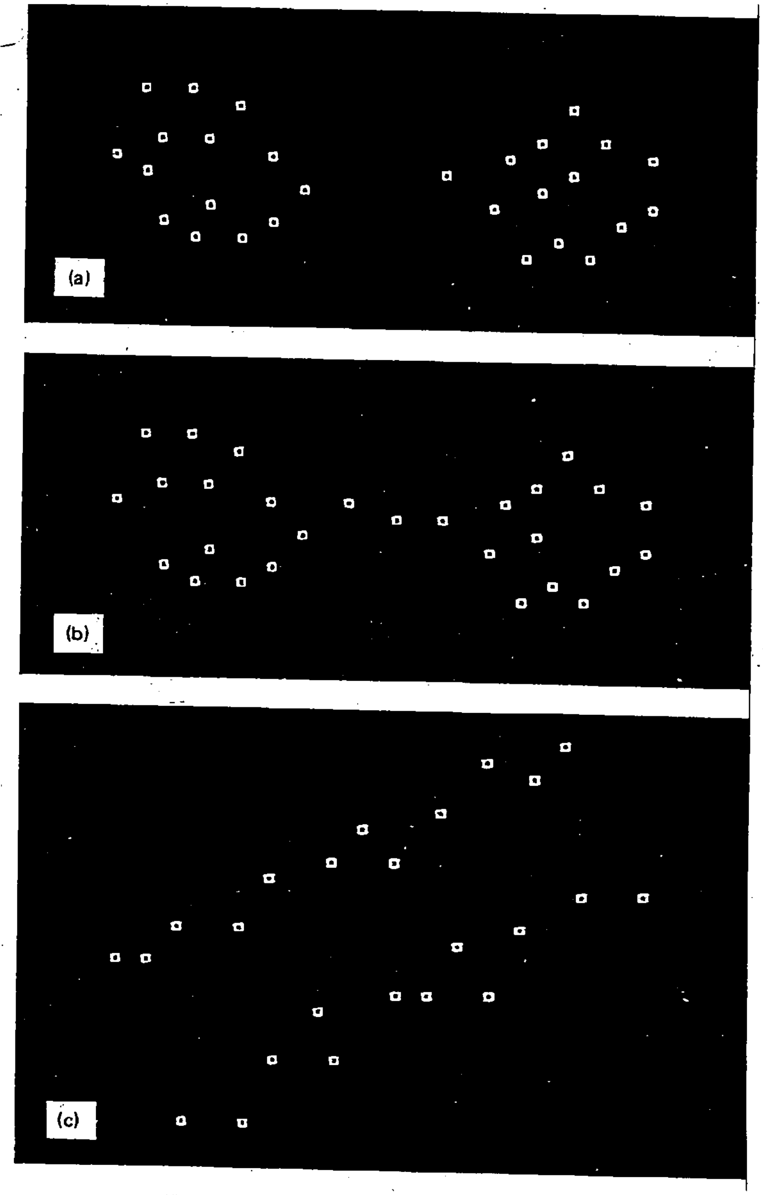


FIGURE 6.16. Three illustrative examples.

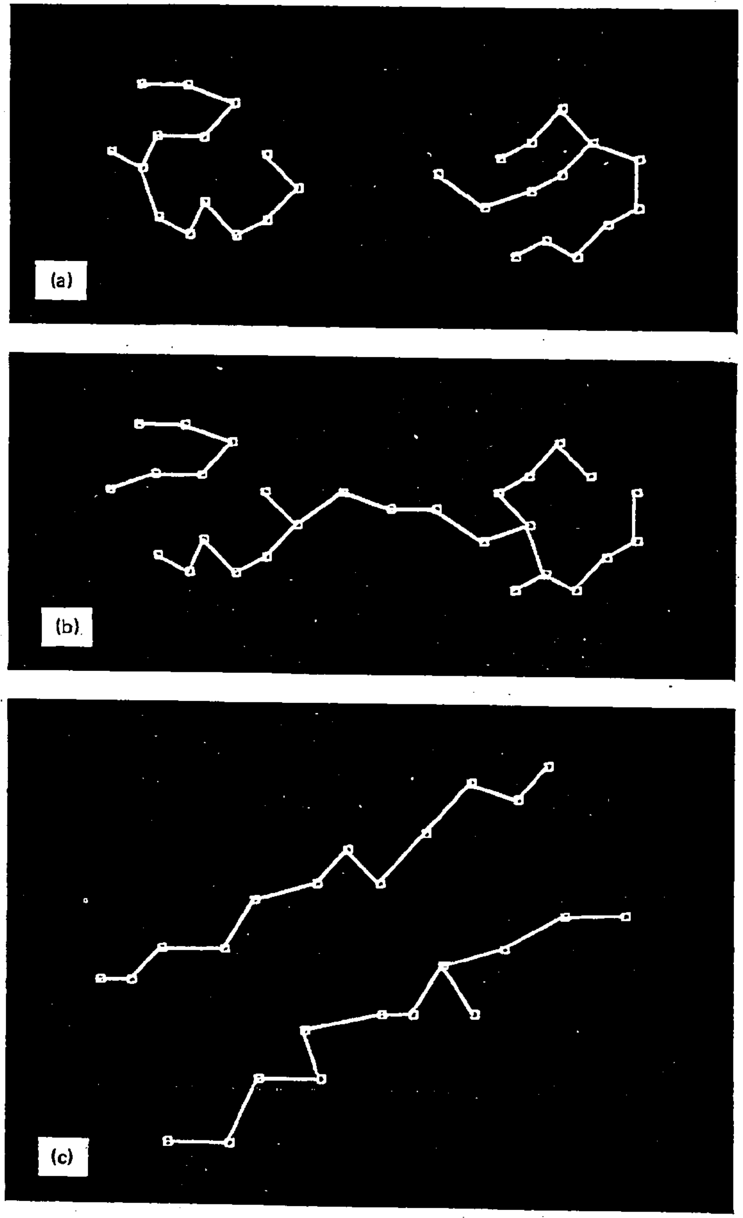


FIGURE 6.17. Results of the nearest-neighbor algorithm.

6.10.2.1 THE NEAREST-NEIGHBOR ALGORITHM

Consider first the behavior when $d_{\min}$ is used.* Suppose that we think of the data points as being nodes of a graph, with edges forming a path between nodes in the same subset $\mathscr{X}_i$.† When $d_{\min}$ is used to measure the distance between subsets, the nearest neighbors determine the nearest subsets. The merging of $\mathscr{X}_i$ and $\mathscr{X}_j$ corresponds to adding an edge between the nearest pair of nodes in $\mathscr{X}_i$ and $\mathscr{X}_j$. Since edges linking clusters always go between distinct clusters, the resulting graph never has any closed loops or circuits; in the terminology of graph theory, this procedure generates a *tree*. If it is allowed to continue until all of the subsets are linked, the result is a *spanning tree*, a tree with a path from any node to any other node. Moreover, it can be shown that the sum of the edge lengths of the resulting tree will not exceed the sum of the edge lengths for any other spanning tree for that set of samples. Thus, with the use of $d_{\min}$ as the distance measure, the agglomerative clustering procedure becomes an algorithm for generating a *minimal spanning tree.*

Figure 6.17 shows the results of applying this procedure to the data of Figure 6.16. In all cases the procedure was stopped at $c = 2$; a minimal spanning tree can be obtained by adding the shortest possible edge between the two clusters. In the first case where the clusters are compact and well separated, the obvious clusters are found. In the second case, the presence of a few points located so as to produce a bridge between the clusters results in a rather unexpected grouping into one large, elongated cluster, and one small, compact cluster. This behavior is often called the "chaining effect," and is sometimes considered to be a defect of this distance measure. To the extent that the results are very sensitive to noise or to slight changes in position of the data points, this is certainly a valid criticism. However, as the third case illustrates, this very tendency to form chains can be advantageous if the clusters are elongated or possess elongated limbs.

6.10.2.2 THE FURTHEST-NEIGHBOR ALGORITHM

When $d_{\max}$ is used to measure the distance between subsets, the growth of elongated clusters is discouraged.‡ Application of the procedure can be thought of as producing a graph in which edges connect all of the nodes in

* In the literature, the resulting procedure is often called the *nearest-neighbor* or the *minimum* algorithm. If it is terminated when the distance between nearest clusters exceeds an arbitrary threshold, it is called the *single-linkage* algorithm.

† Although we will not make deep use of graph theory, we assume that the reader has a general familiarity with the subject. A clear, rigorous treatment is given by O. Ore, *Theory of Graphs* (American Math. Soc. Colloquium Publ., Vol. 38, 1962).

‡ In the literature, the resulting procedure is often called the *furthest neighbor* or the *maximum* algorithm. If it is terminated when the distance between nearest clusters exceeds an arbitrary threshold, it is called the *complete-linkage* algorithm.

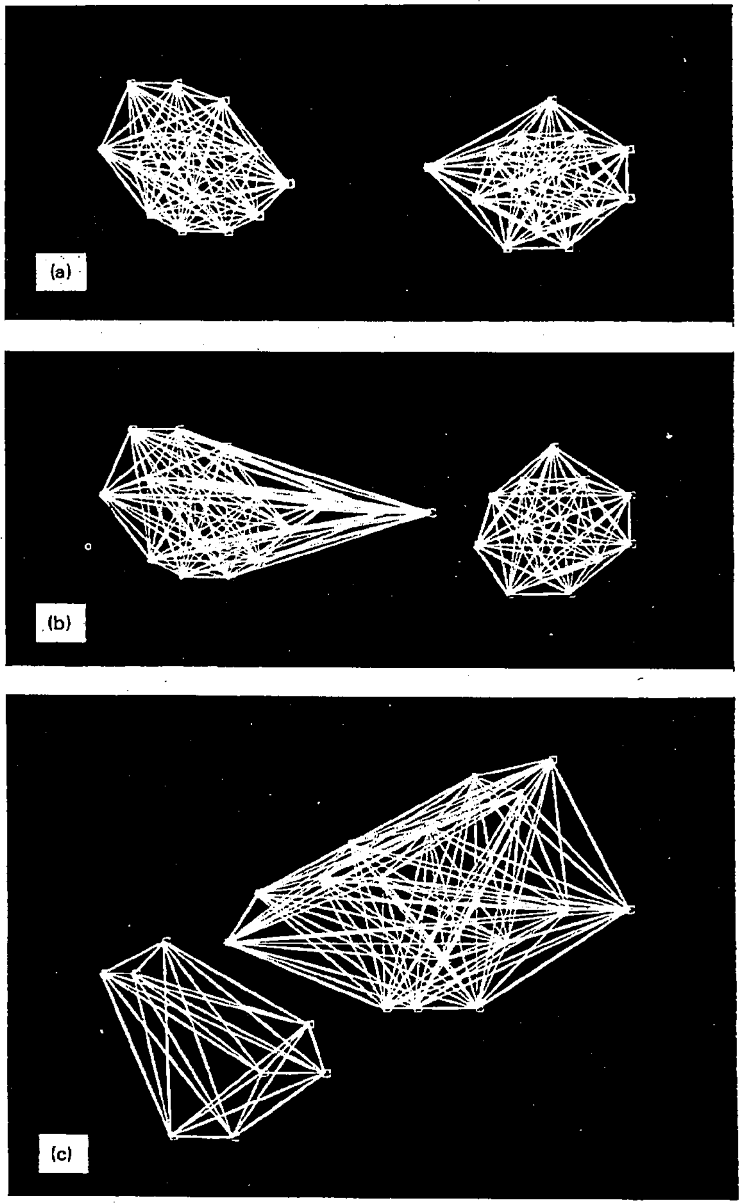


FIGURE 6.18. Results of the furthest-neighbor algorithm.

a cluster. In the terminology of graph theory, every cluster constitutes a *complete* subgraph. The distance between two clusters is determined by the most distant nodes in the two clusters. When the nearest clusters are merged, the graph is changed by adding edges between every pair of nodes in the two clusters. If we define the *diameter* of a cluster as the largest distance between points in the cluster, then the distance between two clusters is merely the diameter of their union. If we define the diameter of a partition as the largest diameter for clusters in the partition, then each iteration increases the diameter of the partition as little as possible. As Figure 6.18 illustrates, this is advantageous when the true clusters are compact and roughly equal in size. However, when this is not the case, as happens with the two elongated clusters, the resulting groupings can be meaningless. This is another example of imposing structure on data rather than finding structure in it.

6.10.2.3 COMPROMISES

The minimum and maximum measures represent two extremes in measuring the distance between clusters. Like all procedures that involve minima or maxima, they tend to be overly sensitive to "mavericks" or "sports" or "outliers" or "wildshots." The use of averaging is an obvious way to ameliorate these problems, and d_{avg} and d_{mean} are natural compromises between d_{min} and d_{max}. Computationally, d_{mean} is the simplest of all of these measures, since the others require computing all $n_i n_j$ pairs of distances $\|\mathbf{x} - \mathbf{x}'\|$. However, a measure such as d_{avg} can be used when the distances $\|\mathbf{x} - \mathbf{x}'\|$ are replaced by similarity measures, where the similarity between mean vectors may be difficult or impossible to define. We leave it to the reader to decide how the use of d_{avg} or d_{mean} might change the way that the points in Figure 6.16 are grouped.

6.10.3 Stepwise-Optimal Hierarchical Clustering

We observed earlier that if clusters are grown by merging the nearest pair of clusters, then the results have a minimum variance flavor. However, when the measure of distance between clusters is chosen arbitrarily, one can rarely assert that the resulting partition extremizes any particular criterion function. In effect, hierarchical clustering defines a cluster as whatever results from applying the clustering procedure. However, with a simple modification it is possible to obtain a stepwise-optimal procedure for extremizing a criterion function. This is done merely by replacing Step 3 of the Basic Agglomerative Clustering Procedure (Section 6.10.2) by

3′. Find the pair of distinct clusters $\mathscr{X}_i$ and $\mathscr{X}_j$ whose merger would increase (or decrease) the criterion function as little as possible.

This assures us that at each iteration we have done the best possible thing, even if it does not guarantee that the final partition is optimal.

We saw earlier that the use of $d_{\max}$ causes the smallest possible stepwise increase in the diameter of the partition. Another simple example is provided by the sum-of-squared-error criterion function J_e. By an analysis very similar to that used in Section 6.9, we find that the pair of clusters whose merger increases J_e as little as possible is the pair for which the "distance"

$$d_e(\mathscr{X}_i, \mathscr{X}_j) = \sqrt{\frac{n_i n_j}{n_i + n_j}}\, \|\mathbf{m}_i - \mathbf{m}_j\|$$

is minimum. Thus, in selecting clusters to be merged, this criterion takes into account the number of samples in each cluster as well as the distance between clusters. In general, the use of d_e tends to favor growth by adding singletons or small clusters to large clusters over merging medium-sized clusters. While the final partition may not minimize J_e, it usually provides a very good starting point for further iterative optimization.

6.10.4 Hierarchical Clustering and Induced Metrics

Suppose that we are unable to supply a metric for our data, but that we can measure a *dissimilarity* value $\delta(\mathbf{x}, \mathbf{x}')$ for every pair of samples, where $\delta(\mathbf{x}, \mathbf{x}') \geq 0$, equality holding if and only if $\mathbf{x} = \mathbf{x}'$. Then agglomerative clustering can still be used, with the understanding that the nearest pair of clusters is the least dissimilar pair. Interestingly enough, if we define the dissimilarity between two clusters by

$$\delta_{\min}(\mathscr{X}_i, \mathscr{X}_j) = \min_{\mathbf{x}\in\mathscr{X}_i, \mathbf{x}'\in\mathscr{X}_j} \delta(\mathbf{x}, \mathbf{x}')$$

or

$$\delta_{\max}(\mathscr{X}_i, \mathscr{X}_j) = \max_{\mathbf{x}\in\mathscr{X}_i, \mathbf{x}'\in\mathscr{X}_j} \delta(\mathbf{x}, \mathbf{x}'),$$

then the hierarchical clustering procedure will induce a distance function for the given set of n samples. Furthermore, the ranking of the distances between samples will be invariant to any monotonic transformation of the dissimilarity values.

To see how this comes about, we begin by defining the *value* v_k for the clustering at level k. For level 1, $v_1 = 0$. For all higher levels, v_k is the minimum dissimilarity between pairs of distinct clusters at level $k - 1$. A moment's reflection will make it clear that with both $\delta_{\min}$ and $\delta_{\max}$ the value v_k either stays the same or increases as k increases. Moreover, we shall assume that no two of the n samples are identical, so that $v_2 > 0$. Thus, $0 = v_1 < v_2 \leq v_3 \leq \cdots \leq v_n$.

We can now define the *distance* $d(\mathbf{x}, \mathbf{x}')$ between $\mathbf{x}$ and $\mathbf{x}'$ as the value of the lowest level clustering for which $\mathbf{x}$ and $\mathbf{x}'$ are in the same cluster. To show that this is a legitimate distance function, or *metric*, we need to show three things:

(1) $d(\mathbf{x}, \mathbf{x}') = 0 \Leftrightarrow \mathbf{x} = \mathbf{x}'$
(2) $d(\mathbf{x}, \mathbf{x}') = d(\mathbf{x}', \mathbf{x})$
(3) $d(\mathbf{x}, \mathbf{x}'') \leq d(\mathbf{x}, \mathbf{x}') + d(\mathbf{x}', \mathbf{x}'')$.

It is easy to see that the first requirement is satisfied. The lowest level for which $\mathbf{x}$ and $\mathbf{x}$ are in the same cluster is level 1, so that $d(\mathbf{x}, \mathbf{x}) = v_1 = 0$. Conversely, if $d(\mathbf{x}, \mathbf{x}') = 0$, the fact that $v_2 > 0$ implies that $\mathbf{x}$ and $\mathbf{x}'$ must be in the same cluster at level 1, and hence that $\mathbf{x} = \mathbf{x}'$. The truth of the second requirement follows immediately from the definition of $d(\mathbf{x}, \mathbf{x}')$. This leaves the third requirement, the triangle inequality. Let $d(\mathbf{x}, \mathbf{x}') = v_i$ and $d(\mathbf{x}', \mathbf{x}'') = v_j$, so that $\mathbf{x}$ and $\mathbf{x}'$ are in the same cluster at level i and $\mathbf{x}'$ and $\mathbf{x}''$ are in the same cluster at level j. Because of the hierarchical nesting of clusters, one of these clusters includes the other. If $k = \max(i, j)$, it is clear that at level k $\mathbf{x}$, $\mathbf{x}'$, and $\mathbf{x}''$ are all in the same cluster, and hence that

$$d(\mathbf{x}, \mathbf{x}'') \leq v_k.$$

But since the values v_k are monotonically nondecreasing, it follows that $v_k = \max(v_i, v_j)$ and hence that

$$d(\mathbf{x}, \mathbf{x}'') \leq \max(d(\mathbf{x}, \mathbf{x}'), d(\mathbf{x}', \mathbf{x}'')).$$

This is known as the *ultrametric inequality*. It is even stronger than the triangle inequality, since $\max(d(\mathbf{x}, \mathbf{x}'), d(\mathbf{x}', \mathbf{x}'')) \leq d(\mathbf{x}, \mathbf{x}') + d(\mathbf{x}', \mathbf{x}'')$. Thus, all the conditions are satisfied, and we have created a bona fide metric for comparing the n samples.

6.11 GRAPH THEORETIC METHODS

In two or three instances we have used linear graphs to add insight into the nature of certain clustering procedures. Where the mathematics of normal mixtures and minimum-variance partitions seems to keep returning us to the picture of clusters as isolated clumps of points, the language and concepts of graph theory lead us to consider much more intricate structures. Unfortunately, few of these possibilities have been systematically explored, and there is no uniform way of posing clustering problems as problems in graph theory. Thus, the effective use of these ideas is still largely an art, and the reader who wants to explore the possibilities should be prepared to be creative.

We begin our brief look into graph-theoretic methods by reconsidering the simple procedure that produced the graphs shown in Figure 6.8. Here a threshold distance d_0 was selected, and two points were said to be in the same cluster if the distance between them was less than d_0. This procedure can easily be generalized to apply to arbitrary similarity measures. Suppose that we pick a threshold value s_0 and say that $\mathbf{x}$ is similar to $\mathbf{x}'$ if $s(\mathbf{x}, \mathbf{x}') > s_0$. This defines an n-by-n *similarity matrix* $S = [s_{ij}]$, where

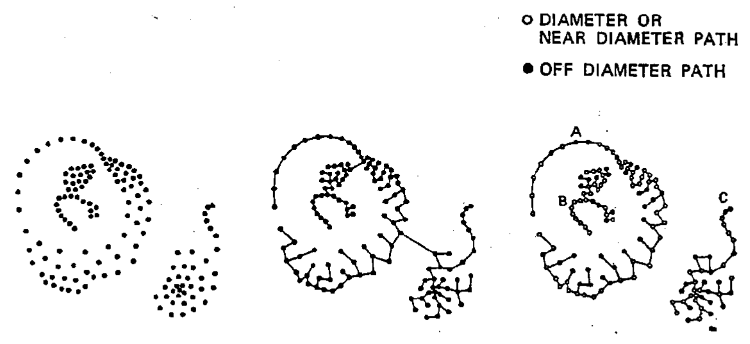
$$s_{ij} = \begin{cases} 1 & \text{if } s(\mathbf{x}_i, \mathbf{x}_j) > s_0 \\ 0 & \text{otherwise} \end{cases} \qquad i, j = 1, \ldots, n$$

This matrix defines a *similarity graph* in which nodes correspond to points and an edge joins node i and node j if and only if $s_{ij} = 1$.

The clusterings produced by the single-linkage algorithm and by a modified version of the complete-linkage algorithm are readily described in terms of this graph. With the single-linkage algorithm, two samples $\mathbf{x}$ and $\mathbf{x}'$ are in the same cluster if and only if there exists a chain $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k$ such that $\mathbf{x}$ is similar to $\mathbf{x}_1$, $\mathbf{x}_1$ is similar to $\mathbf{x}_2$, and so on for the whole chain. Thus, this clustering corresponds to the *connected components* of the similarity graph. With the complete-linkage algorithm, all samples in a given cluster must be similar to one another, and no sample can be in more than one cluster. If we drop this second requirement, then this clustering corresponds to the *maximal complete subgraphs* of the similarity graph, the "largest" subgraphs with edges joining all pairs of nodes. (In general, the clusters of the complete-linkage algorithm will be found among the maximal complete subgraphs, but they cannot be determined without knowing the unquantized similarity values.)

In the preceding section we noted that the nearest-neighbor algorithm could be viewed as an algorithm for finding a minimal spanning tree. Conversely, given a minimal spanning tree we can find the clusterings produced by the nearest-neighbor algorithm. Removal of the longest edge produces the two-cluster grouping, removal of the next longest edge produces the three-cluster grouping, and so on. This amounts to an inverted way of obtaining a divisive hierarchical procedure, and suggests other ways of dividing the graph into subgraphs. For example, in selecting an edge to remove, we can compare its length to the lengths of other edges incident upon its nodes. Let us say that an edge is *inconsistent* if its length l is significantly larger than $\bar{l}$, the average length of all other edges incident on its nodes. Figure 6.19 shows a minimal spanning tree for a two-dimensional point set and the clusters obtained by systematically removing all edges for which $l > 2\bar{l}$. Note how the sensitivity of this criterion to local conditions gives results that are quite different from merely removing the two longest edges.

When the data points are strung out into long chains, a minimal spanning tree forms a natural skeleton for the chain. If we define the *diameter path* as the longest path through the tree, then a chain will be characterized by the shallow depth of branching off the diameter path. In contrast, for a large, uniform cloud of data points, the tree will usually not have an obvious diameter path, but rather several distinct, near-diameter paths. For any of these, an appreciable number of nodes will be off the path. While slight changes in the locations of the data points can cause major rerouting of a minimal spanning tree, they typically have little effect on such statistics.



(a) POINT SET (b) MINIMAL SPANNING TREE (c) CLUSTERS

FIGURE 6.19. Clusters formed by removing inconsistent edges (From C. T. Zahn, 1971. Copyright 1971, Institute of Electrical and Electronics Engineers, reprinted by permission.)

One of the useful statistics that can be obtained from a minimal spanning tree is the edge length distribution. Figure 6.20 shows a situation in which a dense cluster is embedded in a sparse one. The lengths of the edges of the minimal spanning tree exhibit two distinct clusters which would easily be detected by a minimum-variance procedure. By deleting all edges longer than some intermediate value, we can extract the dense cluster as the largest connected component of the remaining graph. While more complicated configurations can not be disposed of this easily, the flexibility of the graph-theoretic approach suggests that it is applicable to a wide variety of clustering problems.

6.12 THE PROBLEM OF VALIDITY

With almost all of the procedures we have considered thus far we have assumed that the number of clusters is known. That is a reasonable assumption if we are upgrading a classifier that has been designed on a small sample set, or if we are tracking slowly time-varying patterns. However, it is a very

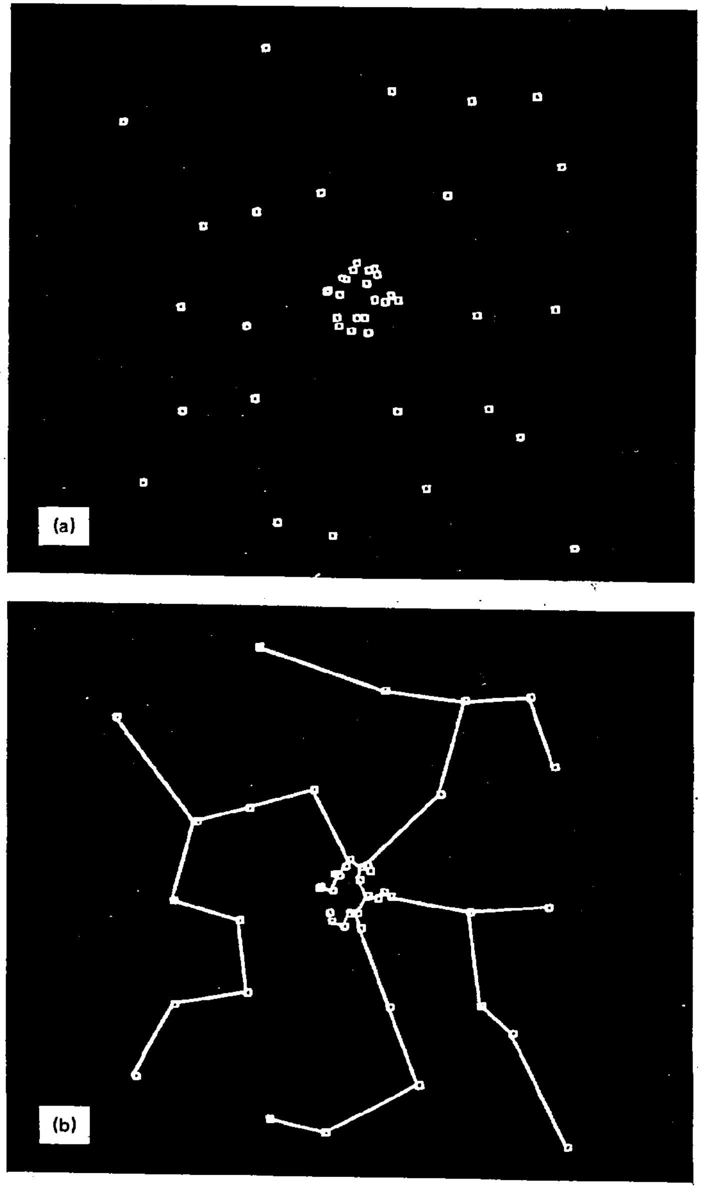


FIGURE 6.20. A minimal spanning tree with a bimodal edge length distribution.

unnatural assumption if we are exploring an essentially unknown set of data. Thus, a constantly recurring problem in cluster analysis is that of deciding just how many clusters are present.

When clustering is done by extremizing a criterion function, a common approach is to repeat the clustering procedure for $c = 1$, $c = 2$, $c = 3$, etc., and to see how the criterion function changes with c. For example, it is clear that the sum-of-squared-error criterion J_e must decrease monotonically with c, since the squared error can be reduced each time c is increased merely by transferring a single sample to the new cluster. If the n samples are really grouped into $\hat{c}$ compact, well separated clusters, one would expect to see J_e decrease rapidly until $c = \hat{c}$, decreasing much more slowly thereafter until it reaches zero at $c = n$. Similar arguments have been advanced for hierarchical clustering procedures, the usual assumption being that large disparities in the levels at which clusters merge indicate the presence of natural groupings.

A more formal approach to this problem is to devise some measure of goodness of fit that expresses how well a given c-cluster description matches the data. The chi-square and Kolmogorov-Smirnov statistics are the traditional measures of goodness of fit, but the curse of dimensionality usually demands the use of simpler measures, such as a criterion function $J(c)$. Since we expect a description in terms of $c + 1$ clusters to give a better fit than a description in terms of c clusters, we would like to know what constitutes a statistically significant improvement in $J(c)$.

A formal way to proceed is to advance the *null hypothesis* that there are exactly c clusters present, and to compute the sampling distribution for $J(c + 1)$ under this hypothesis. This distribution tells us what kind of apparent improvement to expect when a c-cluster description is actually correct. The decision procedure would be to accept the null hypothesis if the observed value of $J(c + 1)$ falls within limits corresponding to an acceptable probability of false rejection.

Unfortunately, it is usually very difficult to do anything more than crudely estimate the sampling distribution of $J(c + 1)$. The resulting solutions are not above suspicion, and the statistical problem of testing cluster validity is still essentially unsolved. However, under the assumption that a suspicious test is better than none, we include the following approximate analysis for the simple sum-of-squared-error criterion.

Suppose that we have a set $\mathscr{X}$ of n samples and we want to decide whether or not there is any justification for assuming that they form more than one cluster. Let us advance the null hypothesis that all n samples come from a normal population with mean μ and covariance matrix $\sigma^2 I$. If this hypothesis were true, any clusters found would have to have been formed by chance, and any observed decrease in the sum of squared error obtained by clustering would have no significance.

The sum of squared error $J_e(1)$ is a random variable, since it depends on the particular set of samples:

$$J_e(1) = \sum_{\mathbf{x}\in\mathscr{X}} \|\mathbf{x} - \mathbf{m}\|^2,$$

where $\mathbf{m}$ is the mean of the n samples. Under the null hypothesis, the distribution for $J_e(1)$ is approximately normal with mean $nd\sigma^2$ and variance $2nd\sigma^4$.

Suppose now that we partition the set of samples into two subsets $\mathscr{X}_1$ and $\mathscr{X}_2$ so as to minimize $J_e(2)$, where

$$J_e(2) = \sum_{i=1}^{2} \sum_{\mathbf{x}\in\mathscr{X}_i} \|\mathbf{x} - \mathbf{m}_i\|^2,$$

$\mathbf{m}_i$ being the mean of the samples in $\mathscr{X}_i$. Under the null hypothesis, this partitioning is spurious, but it nevertheless results in a value for $J_e(2)$ that is smaller than $J_e(1)$. If we knew the sampling distribution for $J_e(2)$, we could determine how small $J_e(2)$ would have to be before we were forced to abandon a one-cluster null hypothesis. Lacking an analytical solution for the optimal partitioning, we cannot derive an exact solution for the sampling distribution. However, we can obtain a rough estimate by considering the suboptimal partition provided by a hyperplane through the sample mean. For large n, it can be shown that the sum of squared error for this partition is approximately normal with mean $n(d - 2/\pi)\sigma^2$ and variance $2n(d - 8/\pi^2)\sigma^4$.

This result agrees with our statement that $J_e(2)$ is smaller than $J_e(1)$, since the mean of $J_e(2)$ for the suboptimal partition—$n(d - 2/\pi)\sigma^2$—is less than the mean for $J_e(1)$—$nd\sigma^2$. To be considered significant, the reduction in the sum of squared error must certainly be greater than this. We can obtain an approximate critical value for $J_e(2)$ by assuming that the suboptimal partition is nearly optimal, by using the normal approximation for the sampling distribution, and by estimating σ^2 by

$$\hat{\sigma}^2 = \frac{1}{nd}\sum_{\mathbf{x}\in\mathscr{X}} \|\mathbf{x} - \mathbf{m}\|^2 = \frac{1}{nd} J_e(1).$$

The final result can be stated as follows: Reject the null hypothesis at the p-percent significance level if

$$\frac{J_e(2)}{J_e(1)} < 1 - \frac{2}{\pi d} - \alpha\sqrt{\frac{2(1 - 8/\pi^2 d)}{nd}}, \tag{44}$$

where α is determined by

$$p = 100\int_{\alpha}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-1/2u^2}\, du.$$

Thus, this provides us with a test for deciding whether or not the splitting of a cluster is justified. Clearly, the c-cluster problem can be treated by applying the same test to all clusters found.

6.13 LOW-DIMENSIONAL REPRESENTATIONS AND MULTIDIMENSIONAL SCALING

Part of the problem of deciding whether or not a given clustering means anything stems from our inability to visualize the structure of multidimensional data. This problem is further aggravated when similarity or dissimilarity measures are used that lack the familiar properties of distance. One way to attack this problem is to try to represent the data points as points in some lower-dimensional space in such a way that the distances between points in the lower-dimensional space correspond to the dissimilarities between points in the original space. If acceptably accurate representations can be found in two or perhaps three dimensions, this can be an extremely valuable way to gain insight into the structure of the data. The general process of finding a configuration of points whose interpoint distances correspond to dissimilarities is often called *multidimensional scaling*.

Let us begin with the simpler case where it is meaningful to talk about the distances between the n samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Let $\mathbf{y}_i$ be the lower-dimensional *image* of $\mathbf{x}_i$, δ_{ij} be the distance between $\mathbf{x}_i$ and $\mathbf{x}_j$, and d_{ij} be the distance between $\mathbf{y}_i$ and $\mathbf{y}_j$. Then we are looking for a *configuration* of image points $\mathbf{y}_1, \ldots, \mathbf{y}_n$ for which the $n(n - 1)/2$ distances d_{ij} between image points are as close as possible to the corresponding original distances δ_{ij}. Since it will usually not be possible to find a configuration for which $d_{ij} = \delta_{ij}$ for all i and j, we need some criterion for deciding whether or not one configuration is better than another. The following sum-of-squared-error functions are all reasonable candidates:

$$J_{ee} = \frac{1}{\sum_{i<j}\delta_{ij}^2}\sum_{i<j}(d_{ij} - \delta_{ij})^2 \tag{45}$$

$$J_{ff} = \sum_{i<j}\left(\frac{d_{ij} - \delta_{ij}}{\delta_{ij}}\right)^2 \tag{46}$$

$$J_{ef} = \frac{1}{\sum_{i<j}\delta_{ij}}\sum_{i<j}\frac{(d_{ij} - \delta_{ij})^2}{\delta_{ij}}. \tag{47}$$

Since these criterion functions involve only the distances between points, they are invariant to rigid-body motions of the configurations. Moreover, they have all been normalized so that their minimum values are invariant to dilations of the sample points. J_{ee} emphasizes the largest errors, regardless whether the distances δ_{ij} are large or small. J_{ff} emphasizes the largest fractional errors, regardless whether the errors $|d_{ij} - \delta_{ij}|$ are large or small. J_{ef} is a useful compromise, emphasizing the largest product of error and fractional error.

Once a criterion function has been selected, an optimal configuration $\mathbf{y}_1, \ldots, \mathbf{y}_n$ is defined as one that minimizes that criterion function. An optimal configuration can be sought by a standard gradient-descent procedure, starting with some initial configuration and changing the $\mathbf{y}_i$'s in the direction of greatest rate of decrease in the criterion function. Since

$$d_{ij} = \|\mathbf{y}_i - \mathbf{y}_j\|,$$

the gradient of d_{ij} with respect to $\mathbf{y}_i$ is merely a unit vector in the direction of $\mathbf{y}_i - \mathbf{y}_j$. Thus, the gradients of the criterion functions are easy to compute:*

$$\nabla_{\mathbf{y}_k} J_{ee} = \frac{2}{\sum\limits_{i<j} \delta_{ij}^2} \sum_{j \neq k} (d_{kj} - \delta_{kj}) \frac{\mathbf{y}_k - \mathbf{y}_j}{d_{kj}}$$

$$\nabla_{\mathbf{y}_k} J_{ff} = 2 \sum_{j \neq k} \frac{d_{kj} - \delta_{kj}}{\delta_{kj}^2} \frac{\mathbf{y}_k - \mathbf{y}_j}{d_{kj}}$$

$$\nabla_{\mathbf{y}_k} J_{ef} = \frac{2}{\sum\limits_{i<j} \delta_{ij}} \sum_{j \neq k} \frac{d_{kj} - \delta_{kj}}{\delta_{kj}} \frac{\mathbf{y}_k - \mathbf{y}_j}{d_{kj}}.$$

The starting configuration can be chosen randomly, or in any convenient way that spreads the image points about. If the image points lie in a $\hat{d}$-dimensional space, then a simple and effective starting configuration can be found by selecting those $\hat{d}$ coordinates of the samples that have the largest variance.

The following example illustrates the kind of results than can be obtained by these techniques.† The data consist of thirty points spaced at unit intervals along a three-dimensional helix:

$$\begin{aligned} x_1(k) &= \cos x_3 \\ x_2(k) &= \sin x_3 \\ x_3(k) &= k/\sqrt{2}, \qquad k = 0, 1, \ldots, 29. \end{aligned}$$

* Second partial derivatives can also be computed easily, so that Newton's algorithm can be used. Note that if $\mathbf{y}_i = \mathbf{y}_j$, the unit vector from $\mathbf{y}_i$ to $\mathbf{y}_j$ is undefined. Should that situation arise, $(\mathbf{y}_i - \mathbf{y}_j)/d_{ij}$ can be replaced by an arbitrary unit vector.

† This example was taken from J. W. Sammon, Jr., "A nonlinear mapping for data structure analysis," *IEEE Trans. Comp.*, C-18, 401–409 (May 1969).

Figure 6.21(a) shows a perspective representation of the three-dimensional data. When the J_{ef} criterion was used, twenty iterations of a gradient descent procedure produced the two-dimensional configuration shown in Figure 6.21(b). Of course, translations, rotations, and reflections of this configuration would be equally good solutions.

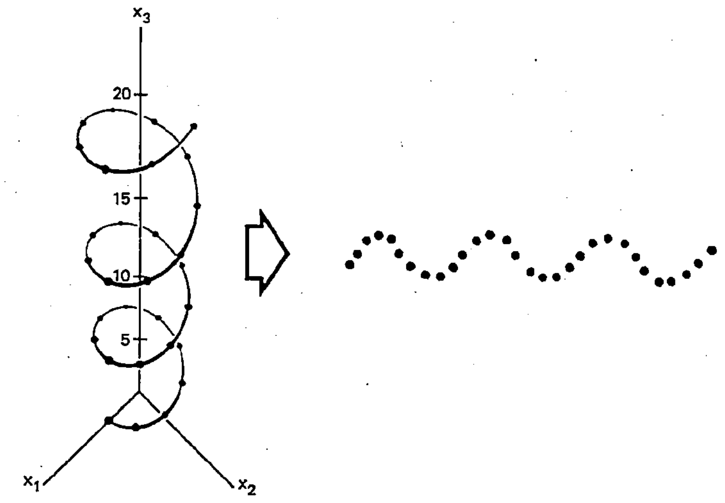
In nonmetric multidimensional scaling problems, the quantities δ_{ij} are dissimilarities whose numerical values are not as important as their rank order. An ideal configuration would be one for which the rank order of the distances d_{ij} is the same as the rank order of the dissimilarities δ_{ij}. Let us order the $m = n(n-1)/2$ dissimilarities so that $\delta_{i_1 j_1} \leq \cdots \leq \delta_{i_m j_m}$, and let $\hat{d}_{ij}$ be *any m* numbers satisfying the *monotonicity constraint*

$$\hat{d}_{i_1 j_1} \leq \hat{d}_{i_2 j_2} \leq \cdots \leq \hat{d}_{i_m j_m}.$$

In general, the distances d_{ij} will not satisfy this constraint, and the numbers $\hat{d}_{ij}$ will not be distances. However, the degree to which the d_{ij} satisfy this constraint is measured by

$$J_{\text{mon}} = \min_{\hat{d}_{ij}} \sum_{i<j} (d_{ij} - \hat{d}_{ij})^2,$$

where it is always to be understood that the $\hat{d}_{ij}$ must satisfy the monotonicity constraint. Thus, J_{mon} measures the degree to which the configuration of



(a) HELIX

(b) IMAGE POINTS

FIGURE 6.21. A two-dimensional representation of data points in three dimensions (Adapted from J. W. Sammon, 1969).

points $\mathbf{y}_1, \ldots, \mathbf{y}_n$ represents the original data. Unfortunately, $\mathcal{J}_{\text{mon}}$ can not be used to define an optimal configuration because it can be made to vanish by collapsing the configuration to a single point. However, this defect is easily removed by a normalization such as the following:

$$J_{\text{mon}} = \frac{\mathcal{J}_{\text{mon}}}{\sum\limits_{i<j} d_{ij}^2}. \tag{48}$$

Thus, J_{mon} is invariant to translations, rotations, and dilations of the configuration, and an optimal configuration can be defined as one that minimizes this criterion function. It has been observed experimentally that when the number of points is larger than dimensionality of the image space, the monotonicity constraint is actually quite confining. This might be expected from the fact that the number of constraints grows as the square of the number of points, and it is the basis for the frequently encountered statement that this procedure allows the recovery of metric information from nonmetric data. The quality of the representation generally improves as the dimensionality of the image space is increased, and it may be necessary to go beyond three dimensions to obtain an acceptably small value of J_{mon}. However, this may be a small price to pay to allow the use of the many clustering procedures available for data points in metric spaces.

6.14 CLUSTERING AND DIMENSIONALITY REDUCTION

Because the curse of dimensionality plagues so many pattern recognition procedures, a variety of methods for dimensionality reduction have been proposed. Unlike the procedures that we have just examined, most of these methods provide a functional mapping, so that one can determine the image of an arbitrary feature vector. The classical procedures of statistics are *principal components analysis* and *factor analysis*, both of which reduce dimensionality by forming linear combinations of the features.* If we think of the problem as one of removing or combining (i.e., grouping) highly correlated features, then it becomes clear that the techniques of clustering

are applicable to this problem. In terms of the *data matrix*, whose n rows are the d-dimensional samples, ordinary clustering can be thought of as a grouping of the rows, with a smaller number of cluster centers being used to represent the data, whereas dimensionality reduction can be though of as a grouping the columns, with combined features being used to represent the data.

Let us consider a simple modification of hierarchical clustering to reduce dimensionality. In place of an n-by-n matrix of distances between samples, we consider a d-by-d *correlation matrix* $R = [\rho_{ij}]$, where the correlation coefficient ρ_{ij} is related to the covariances (or sample covariances) by

$$\rho_{ij} = \frac{\sigma_{ij}}{\sqrt{\sigma_{ii}\sigma_{jj}}}.$$

Since $0 \leq \rho_{ij}^2 \leq 1$, with $\rho_{ij}^2 = 0$ for uncorrelated features and $\rho_{ij}^2 = 1$ for completely correlated features, ρ_{ij}^2 plays the role of a similarity function for features. Two features for which ρ_{ij}^2 is large are clearly good candidates to be merged into one feature, thereby reducing the dimensionality by one. Repetition of this process leads to the following hierarchical procedure:

```
Procedure: Hierarchical Dimensionality Reduction
        1. Let $\hat{d} = d$ and $\mathscr{F}_i = \{x_i\}$, $i = 1, \ldots, d$.
Loop:   2. If $\hat{d} = d'$, stop.
        3. Compute the correlation matrix and find the most correlated
           pair of distinct clusters of features, say $\mathscr{F}_i$ and $\mathscr{F}_j$.
        4. Merge $\mathscr{F}_i$ and $\mathscr{F}_j$, delete $\mathscr{F}_j$, and decrement $\hat{d}$ by one.
        5. Go to Loop.
```

Probably the simplest way to merge two groups of features is just to average them. (This tacitly assumes that the features have been scaled so that their numerical ranges are comparable.) With this definition of a new feature, there is no problem in defining the correlation matrix for groups of features. It is not hard to think of variations on this general theme, but we shall not pursue this topic further.

For the purposes of pattern *classification*, the most serious criticism of all of the approaches to dimensionality reduction that we have mentioned is that they are overly concerned with faithful *representation* of the data. Greatest emphasis is usually placed on those features or groups of features that have the greatest variability. But for classification, we are interested in *discrimination*, not representation. Roughly speaking, the most interesting features are the ones for which the difference in the class means is large relative to the standard deviations, not the ones for which the standard deviations are large. In short, we are interested in something more like the method of multiple discriminant analysis described in Chapter 4.

* The object of principal components analysis (known in the communication theory literature as the Karhunen-Loéve expansion) is to find a lower-dimensional representation that accounts for the variance of the features. The object of factor analysis is to find a lower-dimensional representation that accounts for the correlations among the features. For more information, see M. G. Kendall and A. Stuart, *The Advanced Theory of Statistics, Vol. 3*, Chapter 43 (Hafner, New York, 1966) or H. H. Harman, *Modern Factor Analysis* (University of Chicago Press, Chicago and London, Second Edition, 1967).

There is a growing body of theory on methods of dimensionality reduction for pattern classification. Some of these methods seek to form new features out of linear combinations of old ones. Others seek merely a smaller subset of the original features. A major problem confronting this theory is that the division of pattern recognition into feature extraction followed by classification is theoretically artificial. A completely optimal feature extractor can never be anything but an optimal classifier. It is only when constraints are placed on the classifier or limitations are placed on the size of the set of samples that one can formulate nontrivial (and very complicated) problems. Various ways of circumventing this problem that may be useful under the proper circumstances can be found in the literature, and we have included a few entry points to this literature. When it is possible to exploit knowledge of the problem domain to obtain more informative features, that is usually a more profitable course of action. In the second half of this book we shall devote ourselves to a systematic examination of ways of extracting features from visual data, and with the larger problem of visual scene analysis.

6.15 BIBLIOGRAPHICAL AND HISTORICAL REMARKS

The literature on unsupervised learning and clustering is so large and is scattered across so many disciplines that the following references must be viewed as little more than a selective random sampling. Fortunately, several of the references we cite contain extensive bibliographies, relieving us of many scholarly burdens. Historically, the literature dates back at least to 1894 when Karl Pearson used sample moments to determine the parameters in a mixture of two univariate normal densities. Assuming exact knowledge of values of the mixture density, Doetsch (1936) used Fourier transforms to decompose univariate normal mixtures. Medgyessy (1961) extended this approach to other classes of mixtures, in the process exposing the problem of identifiability. Teicher (1961, 1963) and later Yakowitz and Spragins (1968) demonstrated the identifiability of several families of mixtures, the latter authors showing the equivalence of identifiability and linear independence of the component densities.

The phrases "unsupervised learning" or "learning without a teacher" usually refer to estimation of parameters of the component densities from samples drawn from the mixture density. Spragins (1966) and Cooper (1969) give valuable surveys of this work, and its relation to compound sequential Bayes learning is clarified by Cover (1969). Some of this work is quite general, being primarily concerned with theoretical possibilities. Thus, Stanat (1968) shows how Doetsch's method can be applied to learn multivariate normal and multivariate Bernoulli mixtures, and Yakowitz (1970) demonstrates the possibility of learning virtually any identifiable mixture.

Surprisingly few papers treat maximum-likelihood estimates. Hasselblad (1966) derived maximum-likelihood formulas for estimating the parameters of univariate normal mixtures. Day (1969) derived the formulas for the multivariate, equal covariance matrix case, and pointed out the existence of singular solutions with general normal mixtures. Our treatment of the multivariate case is based directly on the exceptionally clear paper by Wolfe (1970), who also derived formulas for multivariate Bernoulli mixtures. The formulation of the Bayesian approach to unsupervised learning is usually attributed to Daly (1962); more general formulations have since been given by several authors (cf., Hilborn and Lainiotis 1968). Daly pointed out the exponential growth of the optimum system and the need for approximate solutions. Spragins' survey provides references to the literature on decision-directed approximations prior to 1966, with subsequent work being referenced by Patrick, Costello, and Monds (1970). Approximate solutions have also been obtained by the use of histograms (Patrick and Hancock 1966), quantized parameters (Fralick 1967), and randomized decisions (Agrawala 1970).

We have not mentioned all the ways that one might use to estimate unknown parameters. In particular, we have neglected the time-honored and robust method of sample moments, primarily because the situation becomes very complicated when there are more than two components in the mixture. However, some interesting solutions for special cases have been derived by David and Paul Cooper (1964) and elaborated further by Paul Cooper (1967). Because of its slow convergence, we have also omitted mention of the use of stochastic approximation; for the interested reader, the article by Young and Coraluppi (1970) can be recommended.

Much of the early work in clustering was done in the biological sciences, where it appears in studies of numerical taxonomy. Here the major concern is with hierarchical clustering. The influential book by Sokal and Sneath (1963) is an excellent source of references to this literature. Psychologists and sociologists have also contributed to clustering, although they are usually more concerned with clustering features than with clustering samples (Tryon 1939; Tryon and Bailey 1970). The advent of the digital computer made cluster analysis practical, and caused the literature on clustering to spread over many disciplines. The well known survey by Ball (1965) gives a comprehensive overview of this work and is highly recommended; Ball's insights have had a major influence on our treatment of the subject. We have also benefited from the dissertation by Ling (1971), which includes a list of 140 references. The surveys by Bolshev (1969) and Dorofeyuk (1971) give extensive references to the Russian literature on clustering.

Sokal and Sneath (1963) and Ball (1965) list many of the similarity measures and criterion functions that have seen use. The matters of measurement scales, invariance criteria, and appropriate statistical operations are illuminated by Stevens (1968), and related fundamental philosophical issues concerning clustering are treated by Watanabe (1969). The critique of clustering given by Fleiss and Zubin (1969) points out the unhappy consequences of being careless about such matters.

Jones (1968) credits Thorndike (1953) with being the first to use the sum-of-squared-error criterion, which appears so frequently in the literature. The invariant criteria we presented were derived from Friedman and Rubin (1967), who pointed out that these criteria are related to Hotelling's Trace Criterion and the *F*-ratio of classical statistics. The observation that all these criteria give the same optimal partitions in the two-cluster case is due to Fukunaga and Koontz (1970). Of the various criteria we did not mention, the "cohesion" criterion of Watanabe (1969, Chapter 8) is of particular interest since it involves more than pairwise similarity.

In the text we outlined the basic steps in a number of standard optimization and clustering programs. These descriptions were intentionally simplified, and even the more complete descriptions found in the literature do not always mention such matters as how ties are broken or how "wild shots" are rejected. The Isodata algorithm of Ball and Hall (1967) differs from our simplified description in several ways, most notably in the splitting of clusters that have too much within-cluster variability, and the merging of clusters that have too little between-cluster variability. Our description of the basic minimum-squared-error procedure is derived from an unpublished computer program developed by R. C. Singleton and W. H. Kautz at Stanford Research Institute in 1965. This procedure is also closely related to the adaptive sequential procedure of Sebestyen (1962), and to the so-called k-means procedure, whose convergence properties were studied by MacQueen (1967). Interesting applications of such procedures to character recognition are described by Andrews, Atrubin, and Hu (1968) and by Casey and Nagy (1968).

Sokal and Sneath (1963) reference much of the early work on hierarchical clustering, and Wishart (1969) gives explicit references to the original sources for the single-linkage, nearest-neighbor, complete-linkage, furthest-neighbor, minimum-squared-error, and several other procedures. Lance and Williams (1967) show how most of these procedures can be obtained by specializing a general distance function in different ways; in addition, they reference the major papers on divisive hierarchical clustering. The relation between single-linkage procedures and minimal spanning trees was shown by Gower and Ross (1969), who recommended a simple, efficient algorithm for finding minimal spanning trees given by Prim (1957). The equivalence between hierarchical clustering and a distance function satisfying the ultrametric inequality was shown by Johnson (1967).

The great majority of papers on clustering have either explicitly or implicitly accepted some form of minimum-variance criterion. Wishart (1969) pointed out the serious limitations inherent in this approach, and as an alternative suggested a procedure resembling k_n-nearest-neighbor estimation of modes of the mixture density. Critiques of minimum-variance methods have also been given by Ling (1971) and Zahn (1971), both of whom favored graph-theoretic approaches to clustering. Zahn's work, though intended for data of any dimensionality, was motivated by a desire to find mathematical procedures that group sets of points in two dimensions in a way that seems visually natural. (Haralick and Kelly (1969) and Haralick and Dinstein (1971) also treat certain picture processing operations as clustering procedures, a viewpoint that applies to many of the procedures described in Part II of this book.)

Most of the early work on graph-theoretic methods was done for information retrieval purposes. Auguston and Minker (1970) credit Kuhns (1959) with the first application of graph theory to clustering. They give an experimental comparison of several graph-theoretic techniques intended for information retrieval applications, and give many references to work in this domain. It is interesting that among papers with a graph-theoretic orientation we find three that are concerned with statistical tests for cluster validity, viz., those by Bonner (1964), Hartigan (1967), and Ling (1971). Hall, Tepping, and Ball (1971) computed how the sum of squared error varies with the dimensionality of the data and the assumed number of clusters for both uniform and simplex data, and suggested these distributions as useful standards for comparison. Wolfe (1970) suggests a test for cluster validity based on an assumed chi-square distribution for the log-likelihood function.

Green and Carmone (1970), whose valuable monograph on multidimensional scaling contains an extensive bibliography, trace the origins of multidimensional scaling to a paper by Richardson (1938). Recent interest in the topic was stimulated by two developments, nonmetric multidimensional scaling and computer graphics applications. The nonmetric approach originated by Shepard (1962) and extended by Kruskal (1964a) is well suited to many problems in psychology and sociology. The computational aspects of minimizing the criterion J_{mon} subject to a monotonicity constraint are described in detail by Kruskal (1964b). Calvert (1968) used a variation of Shepard's criterion to provide a two-dimensional computer display of multivariate data. The computationally simpler J_{ef} criterion was proposed and used by Sammon (1969) to display data for interactive analysis.

The interest in man-machine systems stems partly from the difficulty of specifying criterion functions and clustering procedures that do what we really want them to do. Mattson and Dammann (1965) were one of the first to suggest a man-machine solution to this problem. The great potential of interactive systems is well described by Ball and Hall (1970) in a paper on their PROMENADE system. Other well-known systems include BC TRY (Tryon and Bailey 1966; 1970), SARF (Stanley, Lendaris, and Nienow 1967), INTERSPACE (Patrick 1969), and OLPARS (Sammon 1970).

Neither automatic nor man-machine systems for pattern recognition can escape the fundamental problems of high-dimensional data. Various procedures have been proposed for reducing the dimensionality, either by selecting the best subset of the available features or by combining the features, usually in a linear fashion. To avoid enormous computational problems, most of these procedures use some criterion other than probability of error in making the selection. For example, Miller (1962) used a tr $S_W^{-1}S_B$ criterion, Lewis (1962) used an entropy criterion, and Marill and Green (1963) used a divergence criterion. In some cases one can bound the probability of error by more easily computed criterion functions, but the final test is always one of actual performance. In the text we restricted our attention to a simple procedure due to King (1967), selecting it primarily because of its close relation to clustering. An excellent presentation of mathematical methods for dimensionality reduction is given by Meisel (1972).

REFERENCES

1. Agrawala, A. K., "Learning with a probabilistic teacher," *IEEE Trans. Info. Theory*, **IT-16**, 373–379 (July 1970).

2. Andrews, D. R., A. J. Atrubin, and K. C. Hu, "The IBM 1975 Optical Page Reader. Part 3: Recognition logic development," *IBM Journal*, **12**, 334–371 (September 1968).

3. Augustson, J. G. and J. Minker, "An analysis of some graph theoretical cluster techniques," *J. ACM*, **17**, 571–588 (October 1970).

4. Ball, G. H., "Data analysis in the social sciences: what about the details?", *Proc. FJCC*, pp. 533–560 (Spartan Books, Washington, D.C., 1965).

5. Ball, G. H. and D. J. Hall, "A clustering technique for summarizing multivariate data," *Behavioral Science*, **12**, 153–155 (March 1967).

6. Ball, G. H. and D. J. Hall, "Some implications of interactive graphic computer systems for data analysis and statistics," *Technometrics*, **12**, 17–31 (February 1970).

7. Bolshev, L. N., "Cluster analysis," *Bulletin, International Statistical Institute*, **43**, 411–425 (1969).

8. Bonner, R. E. "On some clustering techniques," *IBM Journal*, **8**, 22–32 (January 1964).

9. Calvert, T. W., "Projections of multidimensional data for use in man computer graphics," *Proc. FJCC*, pp. 227–231 (Thompson Book Co., Washington, D.C., 1968).

10. Casey, R. G. and G. Nagy, "An autonomous reading machine," *IEEE Trans. Comp.*, **C-17**, 492–503 (May 1968).

11. Cooper, D. B. and P. W. Cooper, "Nonsupervised adaptive signal detection and pattern recognition," *Information and Control*, **7**, 416–444 (September 1964).

12. Cooper, P. W., "Some topics on nonsupervised adaptive detection for multivariate normal distributions," in *Computer and Information Sciences—II*, pp. 123–146, J. T. Tou, ed. (Academic Press, New York, 1967).

13. Cooper, P. W., "Nonsupervised learning in statistical pattern recognition," in *Methodologies of Pattern Recognition*, pp. 97–109, S. Watanabe, ed. (Academic Press, New York, 1969).

14. Cover, T. M., "Learning in pattern recognition," in *Methodologies of Pattern Recognition*, pp. 111–132, S. Watanabe, ed. (Academic Press, New York, 1969).

15. Daly, R. F., "The adaptive binary-detection problem on the real line," Technical Report 2003–3, Stanford University, Stanford, Calif. (February 1962).

16. Day, N. E., "Estimating the components of a mixture of normal distributions," *Biometrika*, **56**, 463–474 (December 1969).

17. Doetsch, G., "Zerlegung einer Funktion in Gausche Fehlerkurven und zeitliche Zuruckverfolgung eines Temperaturzustandes," *Mathematische Zeitschrift*, **41**, 283–318 (1936).

18. Dorofeyuk, A. A., "Automatic classification algorithms (review)," *Automation and Remote Control*, **32**, 1928–1958 (December 1971).

19. Fleiss, J. L. and J. Zubin, "On the methods and theory of clustering," *Multivariate Behavioral Research*, **4**, 235–250 (April 1969).

20. Fralick, S. C., "Learning to recognize patterns without a teacher," *IEEE Trans. Info. Theory*, **IT-13**, 57–64 (January 1967).

21. Friedman, H. P. and J. Rubin, "On some invariant criteria for grouping data," *J. American Statistical Assn.*, **62**, 1159–1178 (December 1967).

22. Fukunaga, K. and W. L. G. Koontz, "A criterion and an algorithm for grouping data," *IEEE Trans. Comp.*, **C-19**, 917–923 (October 1970).

23. Gower, J. C. and G. J. S. Ross, "Minimum spanning trees and single linkage cluster analysis," *Appl. Statistics*, **18**, No. 1, 54–64 (1969).

24. Green, P. E. and F. J. Carmone, *Multidimensional Scaling and Related Techniques in Marketing Analysis* (Allyn and Bacon, Boston, Mass., 1970).

25. Hall, D. J., B. Tepping, and G. H. Ball, "Theoretical and experimental clustering characteristics for multivariate random and structured data," in "Applications of cluster analysis to Bureau of the Census data," Final Report, Contract Cco-9312, SRI Project 7600, Stanford Research Institute, Menlo Park, Calif. (1970).

EXHIBIT D

3

Measures of Central Tendency

In samples, as well as in populations, one generally finds a preponderance of values somewhere around the middle of the range of observed values. The description of this concentration near the middle is an *average*, or a *measure of central tendency* to the statistician. It is also termed a *measure of location*, for it indicates where, along the measurement scale, the sample or population is located.

Various measures of central tendency are useful parameters, in that they describe a property of populations. This chapter discusses the characteristics of these parameters and the sample statistics that are good estimates of them.

3.1 The Arithmetic Mean

The most widely used measure of central tendency is the *arithmetic mean*,* usually referred to simply as the *mean*,† which is the measure most commonly called an "average."

Each measurement in a population may be referred to as an X_i, (read "X sub i") value. Thus, one measurement might be denoted as X_1, another as X_2, another as X_3, and so on. The subscript i might be any integer value up through N, the total number of

*As an adjective, "arithmetic" is pronounced with the accent on the third syllable. In early literature on the subject, the adjective "arithmetical" was employed.

†The term "mean" (the arithmetic mean, as well as the geometric and harmonic means of Section 3.5) dates from ancient Greece (Walker, 1929: 183).

X values in the population.* The mean of the population is denoted by the Greek letter μ (lowercase mu), and is calculated as the sum of all the X_i values divided by the size of the population.

The calculation of the population mean can be abbreviated concisely by the formula

$$\mu = \frac{\sum_{i=1}^{N} X_i}{N}. \tag{3.1}$$

The Greek letter Σ (capital sigma) means "summation"† and $\sum_{i=1}^{N} X$ means "summation of all X_i values from X_1 through X_N." Thus, for example, $\sum_{i=1}^{4} X_i = X_1+X_2+X_3+X_4$ and $\sum_{i=3}^{5} X_i = X_3 + X_4 + X_5$. Since, in statistical computations, summations are nearly always performed over the entire set of X_i values, this book will assume $\sum X_i$ to mean "sum X_i's over all values of i," simply as a matter of printing convenience, and $\mu = \sum X_i/N$ would therefore designate the same calculation as would $\mu = \sum_{i=1}^{N} X_i/N$.

The most efficient, unbiased, and consistent estimate of the population mean, μ, is the sample mean, denoted as $\bar{X}$ (read as "X bar"). Whereas the size of the population (which we generally do not know) is denoted as N, the size of a sample is indicated by n, and $\bar{X}$ is calculated as

$$\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n} \quad \text{or} \quad \bar{X} = \frac{\sum X_i}{n}, \tag{3.2}$$

which is read "the sample mean equals the sum of all measurements in the sample divided by the number of measurements in the sample."‡ Example 3.1 demonstrates the calculation of the sample mean. Note that the mean has the same units of measurement as do the individual observations. The question of how many decimal places should be reported for the mean will be answered at the end of Section 6.3; until then we shall simply record the mean with one more decimal place than the data.

If, as in Example 3.1, a sample contains multiple identical data for several values of the variable, then it may be convenient to record the data in the form of a frequency

*Charles Babbage (1792–1871) was an English mathematician and inventor, who conceived principles used by modern computers—well before the advent of electronics—and who, in 1832, proposed the modern convention of italicizing letters to denote quantities (Cajori, 1929: 2, 6).

†Swiss mathematician Leonhard Euler, in 1755, was the first to use Σ to denote summation (Cajori, 1928: 2, 61).

‡The modern symbols for plus and minus arose in Germany during the 1480s, with Johann Widman the first to use them in print (Cajori, 1928: 222–223). The modern equal sign was invented by English mathematician Robert Recorde, who published it in 1577 along with the first appearance of "+" and "−" in an English work; and it became standard about the start of the eighteenth century (Cajori, 1928: 164, 232–233). Many other symbols were used for mathematical operations, before and after these introductions (e.g., ibid.: 229–245). Using a horizontal line to express division derives from its use, in denoting fractions, by Arabic author Al-Ḥaṣṣâr in the twelfth century, though it was not consitently employed for several more centuries (ibid.: 269, 310). The solidus ("/") was recommended for division by the English writer Augustus De Morgan in 1845 (ibid.: 312–313), and the Swiss author Johann Heinrich Rahn proposed, in 1659, the symbol "÷" which was previously often used by authors as a minus sign (ibid.: 211, 270).

EXAMPLE 3.1 A sample of 24 from a population of butterfly wing lengths.

X_i (in centimeters): 3.3, 3.5, 3.6, 3.6, 3.7, 3.8, 3.8, 3.8, 3.9, 3.9, 3.9, 4.0, 4.0, 4.0, 4.0, 4.1, 4.1, 4.1, 4.2, 4.2, 4.3, 4.3, 4.4, 4.5.

$$\sum X_i = 95.0 \text{ cm}$$

$$n = 24$$

$$\bar{X} = \frac{\sum X_i}{n} = \frac{95.0 \text{ cm}}{24} = 3.96 \text{ cm}$$

table, as in Example 3.2. Then X_i can be said to denote each of k different measurements and f_i can denote the frequency with which that X_i occurs in the sample. The sample mean may then be calculated, using the sums of the products of f_i and X_i, as*

$$\bar{X} = \frac{\sum_{i=1}^{k} f_i X_i}{n}. \tag{3.3}$$

Example 3.2 demonstrates this calculation for the same data as in Example 3.1.

EXAMPLE 3.2 The data from Example 3.1 recorded as a frequency table.

X_l (cm)	f_i	$f_i X_i$ (cm)
3.3	1	3.3
3.4	0	0
3.5	1	3.5
3.6	2	7.2
3.7	1	3.7
3.8	3	11.4
3.9	3	11.7
4.0	4	16.0
4.1	3	12.3
4.2	2	8.4
4.3	2	8.6
4.4	1	4.4
4.5	1	4.5
	$\sum f_i = 24$	$\sum f_i X_i = 95.0$ cm

$$k = 13$$

$$\sum_{i=1}^{k} f_i = n = 24$$

$$\bar{X} = \frac{\sum_{i=1}^{k} f_i X_i}{n} = \frac{95.0 \text{ cm}}{24} = 3.96 \text{ cm}$$

$$\begin{aligned} \text{median} &= 3.95 \text{ cm} + \left(\tfrac{1}{4}\right)(0.1 \text{ cm}) \\ &= 3.95 \text{ cm} + 0.025 \text{ cm} \\ &= 3.975 \text{ cm} \end{aligned}$$

*Denoting the multiplication of two quantities (e.g., a amd b) by their adjacent placement (i.e., ab) derives from very old practices as far back as Hindu manuscripts from the seventh century (Cajori, 1928: 77, 250). Modern usage also includes Gottfried Wilhelm Leibniz's 1698 recommendation of a dot: $a \cdot b$ (ibid.: 267) and William Oughtred's 1631 suggestion of St. Andrew's cross: $a \times b$ (ibid.: 251). The 1659 use of an asterisk-like symbol "$*$" (ibid.: 212–213) did not persist but resurfaced in electronic computer languages of the latter half of the twentieth century.

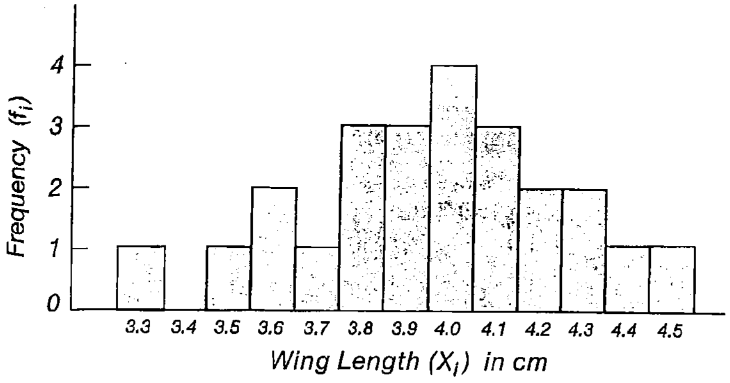


Figure 3.1 A histogram of the data in Example 3.2. The mean (3.96 cm) is the center of gravity of the histogram, and the median (3.975 cm) divides the histogram into two equal areas.

If data are plotted as a histogram (Fig. 3.1), the mean is the *center of gravity* of the histogram.* That is, if the histogram were made of a solid material, it would balance horizontally with the fulcrum at $\bar{X}$. The mean is applicable to ratio or interval scale data.

3.2 The Median

The median is typically defined as the middle measurement in an ordered set of data.† That is, there are just as many observations larger than the median as there are smaller. The sample median is the best estimate of the population median. In a symmetrical distribution (Figs. 3.2a and 3.2b) the sample median is also an unbiased and consistent estimate of μ, but it is not as efficient a statistic as $\bar{X}$, and should not be used as a substitute for $\bar{X}$. (If the frequency distribution is asymmetrical, the median is a poor estimate of the mean.)

The median ($\mathcal{M}$) of a sample of data may be found by first arranging the measurements in order of magnitude. The order may be either ascending or descending, but ascending order is most commonly used as is done with the samples in Examples 3.1, 3.2, and 3.3. Then, we define the sample median as

$$\mathcal{M} = X_{(n+1)/2}. \tag{3.4}$$

If the sample size (n) is odd, then the subscript in Equation 3.4 will be an integer and will indicate which datum is the middle measurement in the ordered sample. For the data of species A in Example 3.3, $n = 9$ and the sample median is $\mathcal{M} = X_{(n+1)/2} = X_{(9+1)/2} = X_5 = 40$ mo. If n is even, then the subscript in Equation 3.4 will be a half-integer, a number midway between two integers. This indicates that there is not a middle value in the ordered list of data; instead, there are two middle values, and the median is defined as the midpoint between them. For the species B data in Example 3.3, $n = 10$ and

*The concept of the mean as the center of gravity was used by L. A. J. Quetelet in 1846 (Walker, 1929: 73).

†The concept of the median was conceived as early as 1816, by K. F. Gauss; enunciated and reinforced by others, including F. Galton in 1869 and 1874; and independently discovered and promoted by G. T. Fechner beginning in 1874 (Walker, 1929: 83–88, 184).

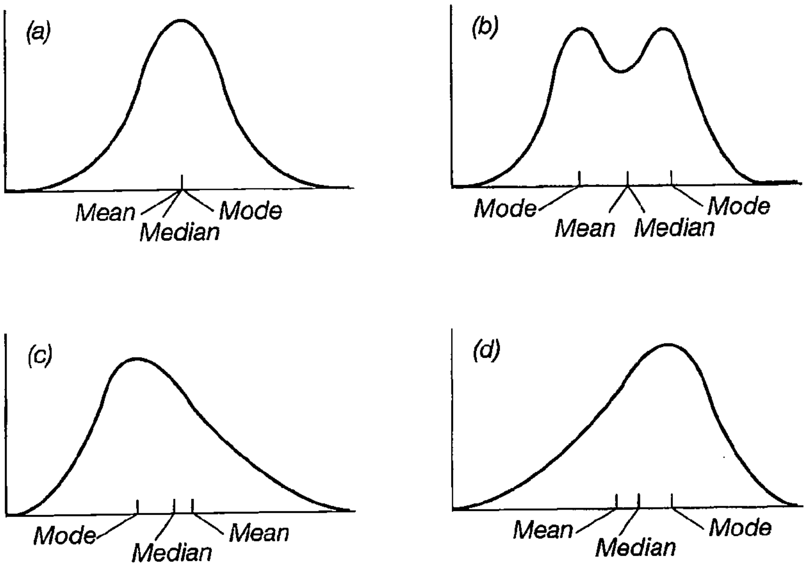


Figure 3.2 Frequency distributions showing measures of central tendency. Values of the variable are along the abscissa (horizontal axis), and the frequencies are along the ordinate (vertical axis). Distributions (a) and (b) are symmetrical, (c) is positively skewed, and (d) is negatively skewed. Distributions (a), (c), and (d) are unimodal, and distribution (b) is bimodal. In a unimodal asymmetric distribution, the median lies about one-third the distance between the mean and the mode.*

EXAMPLE 3.3 Life expectancy for two hypothetical species of birds in captivity.

Species A X_i (mo)	*Species B* X_i (mo)
34	34
36	36
37	37
39	39
40	40
41	41
42	42
43	43
79	44
	45

$n = 9$

$\mathcal{M} = X_{(n+1)/2} = X_{(9+1)/2}$

$= X_5 = 40$ mo

$\bar{X} = 43.4$ mo

$n = 10$

$\mathcal{M} = X_{(n+1)/2} = X_{(10+1)/2}$

$= X_{5.5} = 40.5$ mo

$\bar{X} = 40.1$ mo

*An interesting relationship between the mean, median, and standard deviation is shown in Equation 4.14.

$X_{(n+1)/2} = X_{(10+1)/2} = X_{5.5}$, which signifies that the median is midway between X_5 and X_6, namely $\mathcal{M} = (40\text{ mo} + 41\text{ mo})/2 = 40.5$ mo.

Note that the median has the same units as each individual measurement. If data are plotted as a frequency histogram (e.g., Fig. 3.1), the median is the value of X that divides the area of the histogram into two equal parts. The sample median is, in general, a more efficient estimate of the population median for larger sample sizes.

If we find the middle value(s) in an ordered set of data to be among identical observations (referred to as *tied* values), as in Example 3.1 or 3.2, a difficulty arises. If we apply Equation 3.4 to these twenty-four data, then we conclude the median to be $X_{12.5} = 4.0$ cm. But four data are tied at 4.0 cm, and eleven measurements are less than 4.0 cm and nine are greater. Thus, 4.0 cm does not fit the definition above of the median as that value for which there is the same number of data larger and smaller. Therefore, a better definition of the median of a set of data is that value for which no more than half the data are smaller and no more than half are larger.

When the sample median falls among tied observations, we may interpolate to better estimate the population median. Using the data of Example 3.2, we desire to estimate a value below which 50% of the observations in the population lie. Fifty percent of the observations in the sample would be twelve observations. As the first 7 classes in the frequency table include 11 observations and 4 observations are in class 4.0 cm, we know that the desired sample median lies within the range of 3.95 to 4.05 cm. Assuming that the four observations in class 4.0 cm are distributed evenly within the 0.1 cm range of 3.95 to 4.05 cm, then the median will be $(\frac{1}{4})(0.1\text{ cm}) = 0.025$ cm into this class. Thus, the median = 3.95 cm + 0.025 cm = 3.975 cm. In general, for the sample median within a class interval containing tied observations,

$$\mathcal{M} = \begin{pmatrix}\text{lower limit}\\ \text{of interval}\end{pmatrix} + \left(\frac{0.5n - \text{cum. freq.}}{\text{no. of observations in interval}}\right)\begin{pmatrix}\text{interval}\\ \text{size}\end{pmatrix}, \qquad (3.5)$$

where "cum. freq." refers to the cumulative frequency of the previous classes.* By using this procedure, the calculated median will be the value of X that divides the area of the histogram of the sample into two equal parts. As another example, refer back to Example 1.5, where, by Equation 3.5: median $= \mathcal{M} = 8.75\text{ mg/g} + \{[(0.5)(130) - 61]/24\}\{0.10\text{ mg/g}\} = 8.75\text{ mg/g} + 0.02\text{ mg/g} = 8.77\text{ mg/g}$.

The median expresses less information than does the mean, for it does not take into account the actual value of each measurement, but only considers the rank of each measurement. Still, it may offer certain advantages in some situations. First, it is plain from the two samples in Example 3.3 that extremely high (or extremely low) measurements will not affect the median as much as they affect the mean (causing the sample median to be called a "resistant" statistic). Thus, when we deal with skewed populations, we may prefer the median to the mean to express central tendency.

Note that in Example 3.3 the researcher would have to wait 79 mo to compute a mean life expectancy for species A (45 mo for species B), whereas the median could be

*This procedure was enunciated in 1878 by the German scholar, Gustav Theodor Fechner (1801–1887) (Walker, 1929: 86).

determined in only 40 mo (41 mo for species B). Also, to calculate a median one does not need to have accurate data for all members of the sample. If we did not have the first three data for species A accurately recorded, but could state them as "less than 39 mo," then the median could have been determined just as readily, although calculations of the mean would not have been possible. Lastly, the median can be determined not only for interval and ratio scale data, but also for data on the ordinal scale, data for which the use of the mean usually would not be considered appropriate.

3.3 Other Quantiles

Just as the median is the value above and below which lies half the set of data, one can define measures above or below which lie other fractional parts of the data. For example, if the data are divided into four equal parts, we speak of *quartiles*.

One-fourth of all the ranked observations are smaller than the first quartile, one-fourth lie between the first and second quartiles, one-fourth lie between the second and third quartiles, and one-fourth are larger than the third quartile. The second quartile is identical to the median. As with the median, the first and third quartiles might be one of the data or the midpoint between two of the data. The first quartile, Q_1, is

$$Q_1 = X_{(n+1)/4}; \tag{3.6}$$

if the subscript, $(n+1)/4$, is not an integer or half-integer, then it is rounded up to the nearest integer or half-integer. The second quartile is the median $\mathcal{M}$, and the subscript on X for the third quartile, Q_3, is

$$n + 1 - \text{subscript on } X \text{ for } Q_1. \tag{3.7}$$

Examining the data in Example 3.3: For species $A, n = 9, (n+1)/4 = 2.5$, and $Q_1 = X_{2.5} = 36.5$ mo; and $Q_3 = X_{10-2.5} = X_{7.5} = 42.5$ mo. For species $B, n = 10, (n+1)/4 = 2.75$ (which we round up to 3), and $Q_1 = X_3 = 37$ mo, and $Q_3 = X_{11-3} = X_8 = 43$ mo.

Similarly, values that partition the ordered data set into eight equal parts (or as equal as n will allow) are called *octiles*. The first octile, $\mathcal{O}_1$, is

$$\mathcal{O}_1 = X_{(n+1)/8}; \tag{3.8}$$

and if the subscript, $(n+1)/8$, is not an integer or half-integer, then it is rounded up to the nearest integer or half-integer. The second, fourth, and sixth octiles are the same as the quartiles; i.e., $\mathcal{O}_2 = Q_1$, $\mathcal{O}_4 = Q_2 = \mathcal{M}$, and $\mathcal{O}_6 = Q_3$. The subscript on X for the third octile, $\mathcal{O}_3$, is

$$2(\text{subscript on } X \text{ for } Q_1) - \text{subscript on } X \text{ for } \mathcal{O}_1; \tag{3.9}$$

the subscript on X for the fifth octile, $\mathcal{O}_5$, is

$$n + 1 - \text{subscript on } X \text{ for } \mathcal{O}_3; \tag{3.10}$$

and the subscript on X for the seventh octile, $\mathcal{O}_7$, is

$$n + 1 - \text{subscript on } X \text{ for } \mathcal{O}_1. \tag{3.11}$$

Thus, for the data of Example 3.3: For species A, $n = 9$, $(n+1)/8 = 1.5$ and $\mathcal{O}_1 = X_{1.5} = 35$ mo; $2(2.5) - 1.5 = 3.5$, so $\mathcal{O}_3 = X_{3.5} = 38$ mo; $n + 1 - 3.5 = 6.5$, so $\mathcal{O}_5 = X_{6.5} = 41.5$ mo; and $n + 1 - 1.5 = 8.5$, so $\mathcal{O}_7 = 61$. For species B, $n = 10$, $(n+1)/8 = 1.25$ (which we round up to 1.5) and $\mathcal{O}_1 = X_{1.5} = 35$ mo; $2(3) - 1.5 = 4.5$, so $\mathcal{O}_3 = X_{4.5} = 39.5$ mo; $n+1-4.5 = 6.5$, so $\mathcal{O}_5 = X_{6.5} = 41.5$ mo; and $n + 1 - 1.5 = 9.5$, so $\mathcal{O}_7 = 44.5$ mo.

Besides the median, quartiles, and octiles, ordered data may be divided into fifths, tenths, or hundredths by quantities that are respectively called *quintiles, deciles*, and *centiles* (the latter also called *percentiles*). Measures that divide a group of ordered data into equal parts are collectively termed *quantiles*.* The expression "LD_{50}," used in some areas of biological research, is simply the 50th percentile of the lethal doses, or the median lethal dose. That is, 50% of the experimental subjects survived this dose, whereas 50% did not. Likewise, "LC_{50}" is the median lethal concentration, or the 50th percentile of the lethal concentrations.

3.4 The Mode

The *mode* is commonly defined as the most frequently occurring measurement in a set of data.† In Example 3.2, the mode is 4.0 cm. But it is perhaps better to define a mode as a measurement of relatively great concentration, for some frequency distributions may have more than one such point of concentration, even though these concentrations might not contain precisely the same frequencies. Thus, a sample consisting of the data: 6, 7, 7, 8, 8, 8, 8, 8, 8, 9, 9, 10, 11, 12, 12, 12, 12, 12, 13, 13, and 14 mm would be said to have two modes: at 8 mm and 12 mm. (Some authors would refer to 8 mm as the "major mode" and call 12 mm the "minor mode.") A distribution in which each different measurement occurs with equal frequency is said to have no mode. If two consecutive values of X have frequencies great enough to declare the X values modes, the mode of the distribution is said to be the midpoint of these two X's; e.g., the mode of 3, 5, 7, 7, 7, 8, 8, 8, and 10 liters is 7.5 liters. A distribution with two modes is said to be *bimodal* (e.g., Fig. 3.2b) and may indicate a combination of two distributions with different modes (e.g., heights of men and women). Modes are readily discerned from histograms or frequency polygons.

The sample mode is the best estimate of the population mode. When we sample a symmetrical unimodal population, the mode is an unbiased and consistent estimate of the mean and median (Fig. 3.2a), but it is relatively inefficient and should not be so used.

*Sir Francis Galton developed the concept of percentiles, quartiles, deciles, and other quantiles in writings from 1869 to 1885 (Walker, 1929: 86–87, 177, 179). The term "quantile" was introduced in 1940 by M. G. Kendall (David, 1995).

†The term "mode" was introduced by Karl Pearson in 1894 (Walker, 1929: 184).

As a measure of central tendency, the mode is affected by skewness less than is the mean or the median, but it is more affected by sampling and grouping than these other two measures. The mode, but neither the median nor the mean, may be used for data on the nominal, as well as the ordinal, interval, and ratio scales of measurement. In a unimodal asymmetric distribution (Figs. 3.2c and 3.2d), the median lies about one-third the distance between the mean and the mode.

The mode is not often used in biological research, although it is often interesting to report the number of modes detected in a population, if there are more than one.

3.5 Other Measures of Central Tendency

The *range midpoint*, or *midrange*, is also a measure of central tendency, being half-way between the highest and lowest values in the set of data. It is not to be considered a good estimate of the mean and is a seldom-used measure, for it utilizes relatively little information from the data (although the so-called "mean" daily temperature is often reported as the mean of the minimum and maximum, and is thus a range midpoint). The mean of any two symmetrically located percentiles, such as the mean of the first and third quartiles (i.e., the 25th and 75th percentiles), may be used in the same fashion as the range midpoint as a measure of central tendency (see Dixon and Massey, 1969: 133–134), and it is not as adversely affected by aberrantly extreme values. But such a procedure is seldom encountered. As such measures are based on quantiles, they may be applied to either ratio, interval, or ordinal data.

The *geometric mean* is the nth root* of the product of the n data:

$$\bar{X}_G = \sqrt[n]{X_1 X_2 X_3 \dots X_n} = \sqrt[n]{\prod_{i=1}^{n} X_i}, \tag{3.12}$$

Capital Greek pi, Π, means "take the product" in an analogous fashion as Σ indicates "take the sum." The geometric mean may also be calculated as the antilogarithm of the arithmetic mean of the logarithms of the data (where the logarithms may be in any base); this is much more feasible computationally:

$$\bar{X}_G = \text{antilog}\left(\frac{\log X_1 + \log X_2 + \dots + \log X_n}{n}\right) = \text{antilog}\ \frac{\sum_{i=1}^{n} \log X_i}{n}. \tag{3.13}$$

The geometric mean is appropriate only when all the data are positive. If the data are all equal, then the geometric and arithmetic means are identical; otherwise,† $\bar{X}_G < \bar{X}$. This measure finds use in averaging ratios where it is desired to give each ratio equal weight,

*Denoting the nth root as $\sqrt[n]{\ }$ was suggested by Albert Girard as early as 1629, but this symbol was not generally used until well into the eighteenth century (Cajori, 1928: 371–372).

†The symbols "<" (meaning "less than") and ">" (meaning "greater than") were invented in 1631 by the English writer Thomas Harriot (Cajori, 1928: 199).

and in averaging percent changes, discussions of which are found in Croxton, Cowden, and Klein (1967: 178–182).

The *harmonic mean* is the reciprocal of the arithmetic mean of the reciprocals of the data:

$$\bar{X}_H = \frac{1}{\frac{1}{n}\sum \frac{1}{X_i}} = \frac{n}{\sum \frac{1}{X_i}}. \qquad (3.14)$$

It is occasionally used when dealing with averaging rates, as described by Croxton, Cowden, and Klein (1967: 182–188). If all the data are identical, then the harmonic mean is the same as the arithmetic mean (and also the same as the geometric mean). If they are positive but not identical, then $\bar{X}_H < \bar{X}_G < \bar{X}$.

The geometric and harmonic means are appropriate only for ratio scale data. They are rarely encountered and the term "mean" typically implies "arithmetic mean."

3.6 The Effect of Coding Data

Often in the manipulation of data, considerable time and effort can be saved if *coding* is employed. Coding is the conversion of the original measurements into easier-to-work-with values by simple arithmetic operations. Generally coding employs a *linear transformation* of the data, such as multiplying (or dividing) or adding (or subtracting) a constant. The addition or subtraction of a constant is sometimes termed a translation of the data (i.e., changing the origin), whereas the multiplication or division by a constant causes an expansion or contraction of the scale of measurement. The first set of data in Example 3.4 are coded by subtracting a constant value of 840 g. Not only is each coded value equal to $X_i - 840$ g, but the mean of the coded values is equal to $\bar{X} - 840$ g. Thus, the easier-to-work-with coded values may be used to calculate a mean that then is readily converted to the mean of the original data, simply by adding back the coding constant. In Sample 2 of Example 3.4, the observed data are coded by dividing each observation by 1000 (i.e., by multiplying by 0.001).* The resultant mean only needs to be multiplied by the coding factor of 1000 (i.e., divided by 0.001) to arrive at the mean of the original data. As the other measures of central tendency have the same units as the mean, they are affected by coding in exactly the same fashion. For calculations more involved than computing means, the advantages of coding will become more apparent. In general, linear transformations of ratio or interval scale data will not affect the hypothesis tests to be described later.

In general, if we code X by addition of a constant, A, the coded X is

$$[X_i] = X_i + A. \qquad (3.15)$$

*In 1593, the German astronomer Christoph Clavius (1537–1612) became the first to use a decimal point to separate units from tenths; in 1617, the Scottish mathematician John Napier (1550–1617) used both points and commas for this purpose (Cajori, 1928: 322-323), and the comma is still used in some parts of the world. In some countries a raised dot has been used—a symbol Americans occasinally employ to denote multiplication.

EXAMPLE 3.4 Coding data to facilitate calculations.

Sample 1 (Coding by Subtraction: $A = -840$ g)		Sample 2 (Coding by Division: $M = 0.001$ liters/ml)	
X_i (g)	$[X_i] = X_i - 840$ g	X_i (ml)	$[X_i] = (X_i)(0.001$ liters/ml$)$ $= [X_i]$ liters
842	2	8,000	8.000
844	4	9,000	9.000
846	6	9,500	9.500
846	6	11,000	11.000
847	7	12,500	12.500
848	8	13,000	13.000
849	9		
$\sum X_i = 5922$ g	$\sum [X_i] = 42$ g	$\sum X_i = 63{,}000$ ml	$\sum [X_i] = 63.000$ liters
$\bar{X} = \dfrac{5922 \text{ g}}{7}$ $= 846$ g	$[\bar{X}] = \dfrac{42 \text{ g}}{7}$ $= 6$ g	$\bar{X} = 10{,}500$ ml	$[\bar{X}] = 10.500$ liters

$$\begin{aligned} \bar{X} &= [\bar{X}] - A \\ &= 6 \text{ g} - (-840 \text{ g}) \\ &= 846 \text{ g} \end{aligned}$$

$$\begin{aligned} \bar{X} &= \frac{[\bar{X}]}{M} \\ &= \frac{10.500 \text{ liters}}{0.001 \text{ liters/ml}} \\ &= 10{,}500 \text{ ml} \end{aligned}$$

In Sample 1 of Example 3.4, $A = -840$ g. The mean of a set of data thus coded is

$$[\bar{X}] = \bar{X} + A; \tag{3.16}$$

so if one has calculated $[\bar{X}]$ using coded data, it is a simple matter to determine what the sample mean would have been if the data had not been coded, namely

$$\bar{X} = [\bar{X}] - A. \tag{3.17}$$

If one codes X by multiplying by a constant, M, then each coded datum is

$$[X_i] = MX_i. \tag{3.18}$$

In Sample 2 of Example 3.4, $M = 1/1000 = 0.001$ liters/ml. The mean of the coded data is

$$[\bar{X}] = M\bar{X}. \tag{3.19}$$

Knowing $[\bar{X}]$, one can determine that the mean of the uncoded data is

$$\bar{X} = \frac{[\bar{X}]}{M}. \tag{3.20}$$

Coding affects the median and mode in the same way as the mean is affected.

EXHIBIT E

Contents

Chapter 1 Introduction to Integer Programming

PART I CUTTING PLANE TECHNIQUES

Chapter 2 Dual Fractional Integer Programming

Chapter 12 The Fixed Charge Problem, The Plant Location Problem

Chapter 13 How to do Integer Programming in the Real World

1 Introduction to Integer Programming

1.1 LINEAR PROGRAMS WITH INTEGER VARIABLES

A *linear program* (LP) is a mathematical model which is designed to find a set of nonnegative numbers or variables which maximizes (or minimizes) a linear equation or objective function while satisfying a system of linear constraints. It is apparent that many situations yield linear programming formulations with variables that must have integer values. For example, we cannot build 1.37 schools, manufacture 11.74 aircraft, or award 10.48 (research, production, etc.) contracts. A linear program with some, but not all, of the variables required to be integer, is a *mixed integer (linear) program* (MIP). If all the variables are integer constrained, we have an *integer (linear) program* (IP).

Using matrix notation, a mixed integer program[1] may be written as

$$\begin{array}{lll} \text{(MIP)} & \text{maximize} & cx + dy = z \\ & \text{subject to} & Ax + Dy \leqslant b, \\ & & x \geqslant 0, y \geqslant 0, \\ & \text{and} & x \text{ integer,} \end{array}$$

where

$c = (c_j)$ is an n row (or cost) vector,

$d = (d_j)$ is an n' row (or cost) vector,

$A = (a_{ij})$ is an m by n (constraint) matrix,

[1] We shall interchangeably use MIP, mixed problem, mixed program, mixed integer program, mixed integer problem, and mixed integer linear program. Corresponding equivalences will be used when referring to the integer linear program.

$D=(d_{ij})$ is an m by n' (constraint) matrix,

$b=(b_i)$ is an m column vector of constants (or, simply, the right-hand side),

$x=(x_j)$ is an n vector of integer variables,

and $y=(y_j)$ is an n' vector of continuous variables.

When $n'=0$, the continuous variables y vanish and we have an integer program. If $n=0$, there are no integer variables x and the problem reduces to a linear program.

Mixed integer (or integer) programs in which the integer variables are constrained tǒ be 0 or 1 are called *zero-one mixed integer (or integer) programs*. The following result shows that every mixed (or integer) program in which the integer variables are bounded above can be posed as a zero-one mixed (or integer) problem.

Theorem 1.1 Suppose in problem MIP (or IP) each $x_j \leqslant u_j$ (a positive integer), then the mixed integer (or integer) program is equivalent to a zero-one mixed integer (or integer) program.

PROOF Replace each x_j by either (i) $\Sigma_{k=1}^{u_j} t_{kj}$, where the t_{kj}'s are 0-1 variables, and omit the $x_j \leqslant u_j$ constraints (since $\Sigma_{k=1}^{u_j} t_{kj}$ can at most be u_j), or (ii) $\Sigma_{k=0}^{l_j} 2^k t_{kj}$, where the t_{kj}'s are 0-1 variables and l_j is the smallest integer such that[2] $\Sigma_{k=0}^{l_j} 2^k = 2^{l_j+1}-1 \geqslant u_j$, and retain the $x_j = \Sigma_{k=0}^{l_j} 2^k t_{kj} \leqslant u_j$ constraints. The result follows since either substitution allows x_j to take on any integer value between 0 and u_j. (This is evident in (i) and also in (ii), since every integer can be written in the base two.)■

Example 1.1

If $u_j = 107$, then in (ii), $l_j = 6$, since

$$\sum_{k=0}^{6} 2^k = 2^7 - 1 \geqslant 107 \quad \text{and} \quad \sum_{k=0}^{5} 2^k = 2^6 - 1 < 107;$$

therefore, replace x_j by

$$\sum_{k=0}^{6} 2^k t_{kj} = t_{0j} + 2t_{1j} + 4t_{2j} + 8t_{3j} + 16t_{4j} + 32t_{5j} + 64t_{6j},$$

and retain the constraint $\Sigma_{k=0}^{6} 2^k t_{kj} \leqslant 107$.

Observe that the problem size increases rapidly with u_j. Although (as indicated in Chapters 6 and 7) it is sometimes easier to solve zero-one mixed or integer problems when compared with general ones, the growth in the number of integer variables on such a transformation usually makes it unattractive except perhaps for problems with small values of u_j. Nevertheless, the theorem is of some value since certain results are easier to show for zero-one problems.

[2] The sum of the finite geometric series $a_0 + a_1 + a_2 + \cdots + a_N$ is $(a_0 - a_0 r^{N+1})/(1-r)$, where $r = \frac{a_{i+1}}{a_i}$ $(i = 0, 1, \ldots, N-1)$; and thus the equality.

1.2 USES AND APPLICATIONS

The importance of mixed integer and integer programming is well established. Many mathematical programs can be converted to problems with integer variables (Examples 1.2, 1.3, and 1.4). Also, as already mentioned, many situations yield programming formulations with some or all of the variables required to be integer. Included in these are scheduling, location, network, and selection problems which appear in industry, military, education, health, and other environments. Some classical models are given below (Examples 1.5, 1.6, 1.7, and 1.8), while a more complete discussion may be found in Balinski [3], Balinski and Spielberg [4], and Dantzig [12]. Other applications (mainly of the scheduling and sequencing type which are not described in later chapters) are listed in references [65] through [80]. Some integer programming case studies are included in the final chapter.

1.2.1 Formulations that Allow Integer Variables (Dantzig [12])

Example 1.2 Restrictions on the Solution Values

Suppose a variable z_j is allowed to take on only one of several values, say v_{11}, $v_{12}, \ldots, v_{1N}$. This is equivalent to setting

$$z_j = x_1 v_{11} + x_2 v_{12} + \cdots + x_N v_{1N},$$

with

$$x_1 + x_2 + \cdots + x_N = 1$$

and

$$x_j = 0 \text{ or } 1 \qquad (j = 1, \ldots, N).$$

We may generalize this to the case where the vector $\boldsymbol{Z}$ is allowed to be either the vector $V_1, V_2, \ldots,$ or V_N by setting

$$\boldsymbol{Z} = x_1 V_1 + x_2 V_2 + \cdots + x_N V_N,$$

with

$$x_1 + \cdots + x_N = 1$$

and

$$x_j = 0 \text{ or } 1 \qquad (j = 1, \ldots, N).$$

Example 1.3 Piecewise Convex Constraint Sets

Consider a model whose constraint set consists of several convex subsets. The union of the subsets is not necessarily convex. (For example, they may be disjoint.) To be a solution, a point must satisfy some of the subsets. Specifically, we wish to

$$\begin{aligned} &\text{maximize} && cy \\ &\text{subject to} && b_i - A_i y \geq 0 \qquad (i=1,\ldots,p), \end{aligned}$$

for at least t of the p constraint subsets.[3] Here, A_i is a matrix and b_i is a vector.

This is equivalent to the mixed integer program:

$$\begin{aligned} &\text{maximize} && cy \\ &\text{subject to} && (b_i - A_i y) - x_i L_i \geq 0 \qquad (i=1,\ldots,p), \\ & && \sum_{i=1}^{p} x_i = p - t, \\ &\text{and} && x_i = 0 \text{ or } 1 \qquad (i=1,\ldots,p), \end{aligned}$$

where L_i is a lower bound on the vector function $b_i - A_i y$. When $x_i = 1$, the constraints reduce to $b_i - A_i y \geq L_i$, which are automatically satisfied and may be ignored; if $x_i = 0$, the inequality becomes $b_i - A_i y \geq 0$. So, if $p-t$ of the x_i's are 1, t of the x_i's are 0, and a solution y will satisfy $b_i - A_i y \geq 0$ for at least t of the p constraint sets.

Observe that constraints of the form $(b_i - A_i y) - x_i V_i \leq 0$, where V_i is an upper bound on the vector function $b_i - A_i y$, appear when the constraint subsets are $b_i - A_i y \leq 0$.

As an example consider

$$\begin{aligned} &\text{maximize} && y_1 + y_2 = z \\ &\text{subject to} && \left.\begin{aligned} y_1 &\leq 4 && (1) \\ y_2 &\leq 2 && (2) \end{aligned}\right\} \text{subset 1} \\ & && \left.\begin{aligned} -y_1 + y_2 &\leq 0 && (1)' \\ 4y_1 + 3y_2 &\leq 24 && (2)' \end{aligned}\right\} \text{subset 2} \\ & && y_1, \quad y_2 \geq 0 \end{aligned}$$

for at least one of the two sets (i.e., $t=1$). The (nonconvex) set of feasible solutions appears in Fig. 1.1. Note that the four constraints taken together define the region ***ABCD***, but the optimal solution to this problem occurs at the point ***E***.

To obtain the equivalent mixed integer program we first take the upper bound vector for

$$\begin{pmatrix} y_1 - 4 \\ y_2 - 2 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} -y_1 + y_2 \\ 4y_1 + 3y_2 - 24 \end{pmatrix} \quad \text{to be} \quad \begin{pmatrix} M \\ M \end{pmatrix},$$

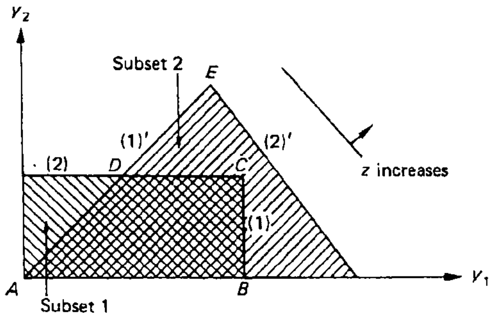


FIGURE 1.1

where M is some large positive number. Then the equivalent problem is

$$\begin{aligned} &\text{maximize} && y_1 + y_2 && = z \\ &\text{subject to} && y_1 - 4 - Mx_1 && \leq 0, \\ & && y_2 - 2 - Mx_1 && \leq 0, \\ & && -y_1 + y_2 - Mx_2 && \leq 0, \\ & && 4y_1 + 3y_2 - 24 - Mx_2 && \leq 0, \\ & && x_1 + x_2 && = 1, \\ &\text{and} && y_1, \quad y_2 \geq 0; \quad x_1, && x_2 = 0 \text{ or } 1. \end{aligned}$$

Example 1.4 Piecewise Linear Objective Functions

Suppose that the objective function z is the sum of piecewise linear functions in one variable; that is, $z = \sum_j f_j(y_j)$, with each $f_j(y_j)$ as in Fig. 1.2. (The subscript j is omitted in the figure.) Then, since any point y in the closed interval $[\bar{y}_i, \bar{y}_{i+1}]$ may be written as $\alpha_i \bar{y}_i + \alpha_{i+1} \bar{y}_{i+1}$, where $\alpha_i + \alpha_{i+1} = 1$ and $\alpha_i, \alpha_{i+1} \geq 0$, we have,

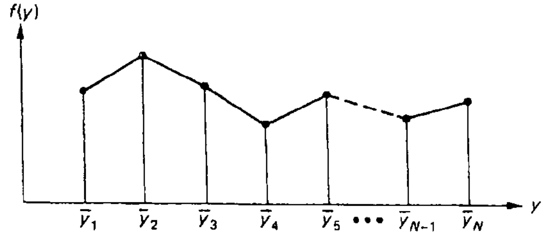


FIGURE 1.2

[3] A linear programming algorithm for the $t=1$ case is described by Fricks [17].

by the linearity of f in the interval, that $f(y)=\alpha_i f(\bar{y}_i)+\alpha_{i+1}f(\bar{y}_{i+1})$. This suggests setting

$$f(y)=\alpha_1 f(\bar{y}_1)+\alpha_2 f(\bar{y}_2)+\cdots+\alpha_N f(\bar{y}_N),$$

where

$$\begin{array}{llllllll}
\alpha_1\bar{y}_1+\alpha_2\bar{y}_2+ & \cdots & + & \alpha_N\bar{y}_N=y & & y\geqslant 0, & \\
\alpha_1 \quad +\alpha_2 \quad + & \cdots & + & \alpha_N \quad =1 & & \alpha_i\geqslant 0 & (i=1,\ldots,N),\\
\alpha_1 & & & & \leqslant x_1, & & \\
\quad \alpha_2 & & & & \leqslant x_1+x_2, & & \\
 & \ddots & & & \vdots & & \\
 & \alpha_{i+1} & & & \leqslant x_i+x_{i+1}, & & \\
 & & \ddots & & \vdots & & \\
 & & & \alpha_N & \leqslant x_{N-1}, & &
\end{array}$$

and

$$x_1+x_2+\cdots+x_{N-1}=1,\quad x_i=0 \text{ or } 1 \qquad (i=1,\ldots,N-1).$$

When an x_i is 1 (with the rest 0), we have $\alpha_i+\alpha_{i+1}=1$ and the other α_i's must be 0, or the program then considers points y in the interval $[\bar{y}_i,\bar{y}_{i+1}]$ and $f(y)$ reduces to $\alpha_i f(\bar{y}_i)+\alpha_{i+1}f(\bar{y}_{i+1})$.

As an example consider

$$\begin{array}{ll}
\text{maximize} & f_1(y_1)+y_2\\
\text{subject to} & y_1 \ +y_2\leqslant 3,\\
 & y_1,\quad y_2\geqslant 0,
\end{array}$$

where

$$f_1(y_1)=\begin{cases} y_1 & \text{for } 0\leqslant y_1\leqslant 2\\ 4-y_1 & \text{for } 2\leqslant y_1\leqslant 3\\ 0 & \text{otherwise.}\end{cases}$$

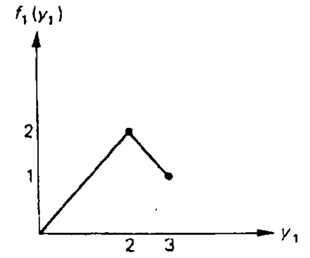


FIGURE 1.3

Then (see Fig. 1.3) $f_1(y_1)=\alpha_1 f_1(0)+\alpha_2 f_1(2)+\alpha_3 f_1(3)=2\alpha_2+\alpha_3$, and $\alpha_1\bar{y}_1+\alpha_2\bar{y}_2+\alpha_3\bar{y}_3=\alpha_1(0)+\alpha_2(2)+\alpha_3(3)=y_1$, or the equivalent problem is

$$\begin{array}{lllll}
\text{maximize} & 2\alpha_2+ \ \alpha_3+y_2 & & & \\
\text{subject to} & 2\alpha_2+ \ 3\alpha_3+y_2 & \leqslant 3, & & \\
 & \alpha_1 & \leqslant x_1, & & \\
 & \quad \alpha_2 & \leqslant x_1+x_2, & & \\
 & \qquad \alpha_3 & \leqslant x_2, & & \\
 & \alpha_1+\ \alpha_2+\ \alpha_3 & =1, & & \\
 & y_1=2\alpha_2+3\alpha_3 & \geqslant 0, & & \\
 & x_1+\ x_2=1, & x_1,x_2=0 \text{ or } 1, & & \\
\text{and} & \alpha_1,\quad \alpha_2,\quad \alpha_3,\quad y_2 & \geqslant 0. & &
\end{array}$$

The transformation given here can be used to approximate a nonlinear function. A nonlinear function can also be linearized by first transforming it into a polynomial function with zero-one variables (Balas [1], Hammer and Rudeanu [37]) and then transforming the polynomial function into a linear function with zero-one variables (Balas [1], Watters [60], Zangwill [64]). However, as suggested by the example, these transformations tend to dramatically increase the number of variables and constraints. Methods to achieve more economical linear representations of zero-one polynomial programming problems are discussed by Glover [23], and Glover and Woolsey [26], [27]. A computational study by Taha [55] indicates that the benefit (in terms of ease of solution) gained by using these transformations is data dependent, and conversion may not be worthwhile. Related work includes the relationship between a zero-one integer program and a quadratic programming problem (Problem 1.4) as discussed in Bowman and Glover [7], Kennington and Fyffe [44], and Raghavachari [51], [52]; other transformations are in Garfinkel and Nemhauser [18], Granot and Hammer [35], and Petersen [50].

1.2.2 Classical Applications

It turns out that a few integer programming models may represent a vast number of real world problems, and as a result they have been titled and considerable work has been done on developing algorithms and computer packages for them (a listing is in Salkin [54]). We now describe these models. Applications and algorithms are left to Chapters 10, 11, 12, and 13.

Example 1.5 Facility Location Problems

In the simplest case there are m sources (or facility locations) which produce a single commodity for n customers each with a demand for d_j units ($j=1,\ldots,n$). If a particular source i is operating (or facility is built), it has a fixed cost $f_i\geqslant 0$

and a production capacity $M_i > 0$ associated with it. There is also a positive cost g_{ij} for shipping a unit from source i to customer j. The question is where to locate the sources so that capacities are not exceeded and demands are met, all at a minimal total cost. All data are assumed to be integral.

To model this problem, we let z_{ij} be the amount shipped from source i to customer j, and define x_i to be 1 if source i is used and 0 if it is not. Then the integer programming model is:

$$\begin{array}{llll}
\text{minimize} & \sum_{i=1}^{m}\sum_{j=1}^{n} g_{ij}z_{ij} + \sum_{i=1}^{m} f_i x_i & & (1) \\
\text{subject to} & \sum_{i=1}^{m} z_{ij} = d_j & (j=1,\ldots,n), & (2) \\
 & \sum_{j=1}^{n} z_{ij} \leq M_i x_i & (i=1,\ldots,m), & (3) \\
 & z_{ij} \geq 0 & (\text{all } i,j), & (4) \\
\text{and} & x_i = 0 \text{ or } 1 & (i=1,\ldots,m). & (5)
\end{array}$$

The objective function (1) is the total shipping cost, i.e., $\Sigma_i\Sigma_j g_{ij}z_{ij}$, plus the total fixed cost, i.e., $\Sigma_i f_i x_i$; note that f_i contributes to this sum only when $x_i = 1$ or source i is used. Constraints (2) guarantee that each customer's demand is met. Inequality (3) ensures that we do not ship from a source which is not operating[4] (M_i is an upper bound on the amount that may be shipped from source i) and it also restricts production from exceeding capacity. Even though by definition z_{ij} is discrete, we may use the nonnegativity conditions (4) because it can be shown that constraints (5) with (2) and (3) ensure that $z_{ij} \geq 0$ will mean that z_{ij} is integer in the optimal solution (Problem 1.5).

In certain situations, because of available equipment or expected demand, it may be reasonable to assume that the amount produced at each facility i will never exceed the capacity M_i. (The resulting problem is sometimes referred to as the *uncapacitated plant location problem.*) Under this assumption, for any fixed zero-one value of x the shipping cost is minimized if each customer's demand d_j is satisfied from the open plant i (i.e., $x_i = 1$) with minimal cost g_{ij}. To illustrate, consider Fig. 1.4, where plants 1 and 2 are open to satisfy the demands of three customers. Also, suppose $g_{11} < g_{21}$, $g_{13} < g_{23}$, and $g_{12} > g_{22}$. Or, to minimize the total shipping cost, we meet the demands of customers 1 and 3 from plant 1 and customer 2 from plant 2. (Alternate optima exist when there are equal shipping costs. For example, if $g_{11} = g_{21}$, minimal solutions contain $z_{11} = K$ and $z_{21} = d_1 - K$ where $0 \leq K \leq d_1$. However, there is always an optimal solution with d_j units obtained from one plant.)

[4] This can also be guaranteed by using constraints of the form $z_{ij} \leq M_i x_i$ for all i, j.

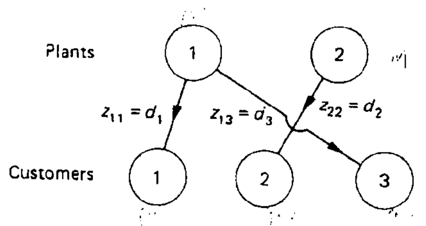


FIGURE 1.4

The above discussion suggests a reformulation of the problem. Suppose we denote the fraction of demand d_j satisfied by plant i as y_{ij}, i.e., $y_{ij} = z_{ij}/d_j$. Then for any x there is a minimal solution with $y_{ij} = 0$ or 1 for all i, j. Further, since the set of inequalities (3) now has the sole function of prohibiting shipments from closed plants and allowing it otherwise, (3) may be replaced by $\Sigma_{j=1}^{n} y_{ij} \leq nx_i$. Letting $c_{ij} = g_{ij}d_j$ yields the reformulated problem (where constraints (2)′ and (4)′ ensure that $y_{ij} \leq 1$)

$$\begin{array}{llll}
\text{minimize} & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}y_{ij} + \sum_{i=1}^{m} f_i x_i & & (1)' \\
\text{subject to} & \sum_{i=1}^{m} y_{ij} = 1 & (j=1,\ldots,n), & (2)' \\
 & \sum_{j=1}^{n} y_{ij} \leq nx_i & (i=1,\ldots,m), & (3)' \\
 & y_{ij} \geq 0 & (\text{all } i,j), & (4)' \\
\text{and} & x_i = 0 \text{ or } 1 & (i=1,\ldots,m). & (5)
\end{array}$$

Example 1.6 Resource–Task Scheduling

In this situation we have n resources which must perform m tasks where each resource can perform some or all of the tasks. For each resource j $(j = 1,\ldots,n)$ we may list, subject to time, location, and other constraints, all possible ways a resource can do the tasks. In particular, for each resource j a zero-one matrix can be developed where the rows correspond to the tasks, the columns to the possible combinations, and an entry is 1(0) if in that combination resource j can (cannot) do the task corresponding to the row. Suppose also that a cost c_{ij} of using the ith possible combination of resource j can be computed (Fig. 1.5). Then we may model the problem by defining the binary variable x_{ij} to be 1 if the ith combination of resource j is used, and $x_{ij} = 0$ otherwise. To ensure that each task is done at least once,

$$e^t x \geq 1 \qquad \text{for each task} \quad t = 1,\ldots,m, \tag{6}$$

where e^t is the row of the binary matrix corresponding to task t and x is a vector

Variables	$(x_{11}$	x_{21}	x_{31}	$\dots$	x_{12}	x_{22}	$\dots$	$\dots$	x_{1n}	$\dots)$	$=x$
Costs	$(c_{11}$	c_{21}	c_{31}	$\dots$	c_{12}	c_{22}	$\dots$	$\dots$	c_{1n}	$\dots)$	$=c$
Resource			1			2		$\dots$	N		
Combination	1	2	3	$\dots$	1	2	$\dots$		1	$\dots$	
Tasks											
(e^1) 1	1	0	0	$\dots$	0	0	$\dots$		1	$\dots$	$\geqslant 1$
(e^2) 2	0	1	0	$\dots$	1	0	$\dots$		1	$\dots$	$\geqslant 1$
$\cdot$ $\cdot$	$\cdot$	$\cdot$	$\cdot$		$\cdot$	$\cdot$			$\cdot$		$\cdot$
$\cdot$ $\cdot$	$\cdot$	$\cdot$	$\cdot$		$\cdot$	$\cdot$			$\cdot$		$\cdot$
$\cdot$ $\cdot$	$\cdot$	$\cdot$	$\cdot$		$\cdot$	$\cdot$			$\cdot$		$\cdot$
(e^m) m	1	1	1	$\dots$	0	1	$\dots$		1	$\dots$	$\geqslant 1$

E

FIGURE 1.5

whose components are the corresponding binary variables x_{ij} (Fig. 1.5). Note that the m inequalities (6) are equivalent to

$$Ex \geqslant e, \tag{7}$$

where E is a zero-one matrix whose rows are e^t and e is an m column of ones. As we wish to minimize the total cost, the objective function is

$$\text{minimize } cx, \tag{8}$$

where c is a vector whose elements are the corresponding costs c_{ij} (Fig. 1.5). Combining (7) and (8) we have the integer program:

$$\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Ex \geqslant e \\ \text{and} & x \text{ binary.} \end{array}$$

This problem is often referred to as a *set covering problem.*

It is often necessary that each task be done exactly once. In this case, the constraints (7) are $Ex = e$, and the model is the *set partitioning problem*:

$$\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Ex = e \\ \text{and} & x \text{ binary.} \end{array}$$

Example 1.7 A Loading Problem

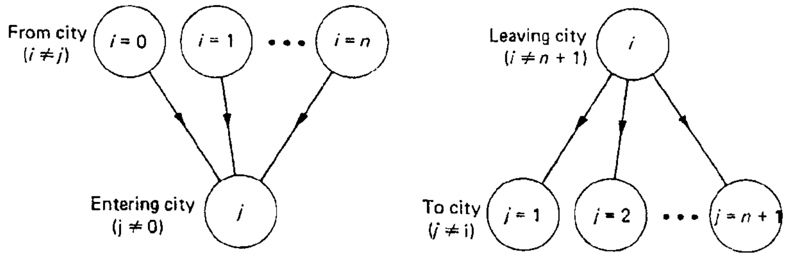
Suppose that a relief plane has cargo weight capacity b and is to be loaded with items each with weight a_j and relative (nonnegative) value c_j. The problem is to load the plane so as to maximize its total relative value. The situation is described by the integer program with a single constraint:

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_j x_j \leqslant b & \\ \text{and} & x_j = 0 \text{ or } 1 & (j = 1, \dots, n), \end{array}$$

where $x_j = 1$ if item j is loaded on the plane and $x_j = 0$ if not. The problem is usually called a *knapsack problem.* It relates to a hiker who can carry at most b pounds and is to select items of weight a_j and value c_j so as to maximize the value of the knapsack.

Example 1.8 The Traveling Salesman Problem

A traveling salesman must visit n cities, each exactly once. The distance between every pair of cities ij, denoted by d_{ij} $(i \neq j)$, is known and may depend on the direction travelled (i.e., d_{ij} does not necessarily equal d_{ji}). The problem is to find a tour which commences and terminates at the salesman's home city and minimizes the total distance traveled.



(a) (b)

FIGURE 1.6

Suppose we label the home city as city 0 *and* as city $n+1$. (Then we may think of the salesman's initial location as city 0 and the desired final location as city $n+1$.) Also, introduce the zero-one variable x_{ij} $(i = 0, 1, \dots, n, j = 1, \dots, n+1, i \neq j)$, where $x_{ij} = 1$ if the salesman travels from city i to j, and $x_{ij} = 0$ otherwise. To guarantee that each city (except city 0) is entered exactly once, we have (Fig. 1.6(a))

$$\sum_{i=0}^{n} x_{ij} = 1 \qquad (j = 1, \dots, n+1, i \neq j).$$

Similarly, to ensure that each city (except city $n+1$) is left exactly once, we have (Fig. 1.6(b))

$$\sum_{j=1}^{n+1} x_{ij}=1 \qquad (i=0,1,\ldots,n,\ i\neq j).$$

These constraints, however, do not eliminate the possibility of subtours or "loops" as indicated by Fig. 1.7, where the solution is $x_{01}=x_{12}=x_{26}=1$, $x_{34}=x_{45}=x_{53}=1$, and $x_{ij}=0$ otherwise (with $n=5$). One way of eliminating the subtour possibility is to add the constraints

$$\alpha_i-\alpha_j+(n+1)x_{ij}\leqslant n \qquad (i=0,1,\ldots,n,\ j=1,\ldots,n+1,\ i\neq j),$$

where α_i is a real number associated with city i.

To show that a solution containing loops cannot satisfy these constraints, consider any subtour except the one containing the home city. Then, if we sum the inequalities corresponding to the $x_{ij}=1$ around the loop, the $\alpha_i-\alpha_j$ cancel each other and we are left with $(n+1)N\leqslant nN$, where N is the number of arcs in the subtour, which is a contradiction. (Relating to Fig. 1.7 and loop 3→4→5, the constraints are $\alpha_3-\alpha_4+6\leqslant 5$, $\alpha_4-\alpha_5+6\leqslant 5$, and $\alpha_5-\alpha_3+6\leqslant 5$; or, after adding, $18\leqslant 15$.)

On the other hand, to see that these constraints may be satisfied when there are no subtours, define $\alpha_0=0$, $\alpha_{n+1}=n+1$, and let $\alpha_i=K$ if city i is the Kth city visited on the tour. Then, when $x_{ij}=1$, we have $\alpha_i-\alpha_j+(n+1)=K-(K+1)+(n+1)=n$. Also, since $1\leqslant\alpha_i\leqslant n+1$ $(i=1,\ldots,n+1)$, the difference $\alpha_i-\alpha_j$ is always $\leqslant n$ (all i,j), and thus the constraints are satisfied when $x_{ij}=0$.

To complete the model we wish to minimize the total distance $\Sigma_{i=0}^{n}\Sigma_{j=1,j\neq i}^{n+1}d_{ij}x_{ij}$. Or an integer programming formulation of the traveling salesman problem is to find variables x_{ij} and arbitrary real numbers α_i which

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{i=0}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n+1} d_{ij}x_{ij} & \\
\text{subject to} & \displaystyle\sum_{i=0}^{n} x_{ij}=1 & (j=1,\ldots,n+1,\ i\neq j),\\
 & \displaystyle\sum_{j=1}^{n+1} x_{ij}=1 & (i=0,1,\ldots,n,\ i\neq j),\\
 & \alpha_i-\alpha_j+(n+1)x_{ij}\leqslant n & (i=0,1,\ldots,n,\ j=1,\ldots,n+1,\ i\neq j),\\
\text{and} & x_{ij}=0 \text{ or } 1 & (i=0,1,\ldots,n,\ j=1,\ldots,n+1,\ i\neq j),
\end{array}$$

where $x_{0,n+1}=0$ (since $x_{ij}=0$ for $i=j$). This formulation originally appeared in Tucker [58].

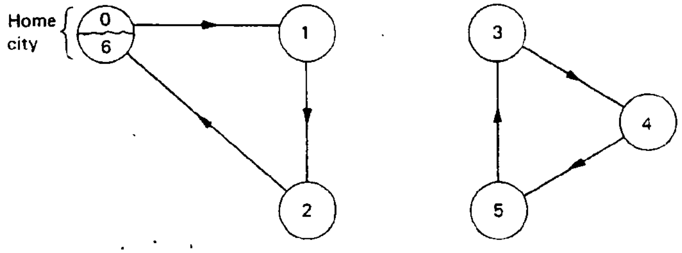


FIGURE 1.7

1.3 GRAPHICAL SOLUTIONS, ROUNDING

From linear programming (Dantzig [13], Hadley [36]) we know that the intersection of the hyperplanes corresponding to the linear constraints $\boldsymbol{Ax}+\boldsymbol{By}\leqslant\boldsymbol{b}$, $\boldsymbol{x}\geqslant\boldsymbol{0}$, and $\boldsymbol{y}\geqslant\boldsymbol{0}$ is a convex polyhedron with a maximum value of the objective function $cx+dy=z$ occurring at one of its extreme points. In general, however, the x variables at such a point will not take on integer values. Since the polyhedron contains *all* solutions with x integer we may (graphically) solve the mixed integer problem by first finding an optimal extreme point of its linear counterpart (e.g., by the simplex method) and then parallel shift the hyperplane $\boldsymbol{cx}+\boldsymbol{dy}=z$ back into the polyhedron until it intersects the first point (x,y) with x integer. Such a solution is optimal. The procedure allows us to solve problems with at most three variables graphically.

Example 1.9 Simple Graphical Solutions

$$\begin{array}{llr}
\text{Maximize} & 2x-\ y=z & \\
\text{subject to} & 5x+7y\leqslant 45, & (9)\\
 & -2x+\ y\leqslant 1, & (10)\\
 & 2x-5y\leqslant 5, & (11)\\
 & x,\quad y\geqslant 0, & (12)\\
\text{and} & x\ \ \text{integer.} &
\end{array}$$

The linear programming feasible region, defined by the constraints (9), (10), (11), and (12), is the convex polyhedron spanned by the extreme points $\boldsymbol{ABCDE}$ (Fig. 1.8). The maximal value of z is 35/3 and occurs at the extreme point $\boldsymbol{C}=(y,x)=(5/3,20/3)$. The set of feasible solutions with x integer is made up of the points on the line segments $l_0,l_1,l_2,l_3,l_4,l_5,l_6$. If the z hyperplane is pushed back into the polyhedron the first point reached with x integer is $\boldsymbol{F}=(7/5,6)$ when $z=53/5$ (Fig. 1.8). If y is also required to be integer, solutions to the

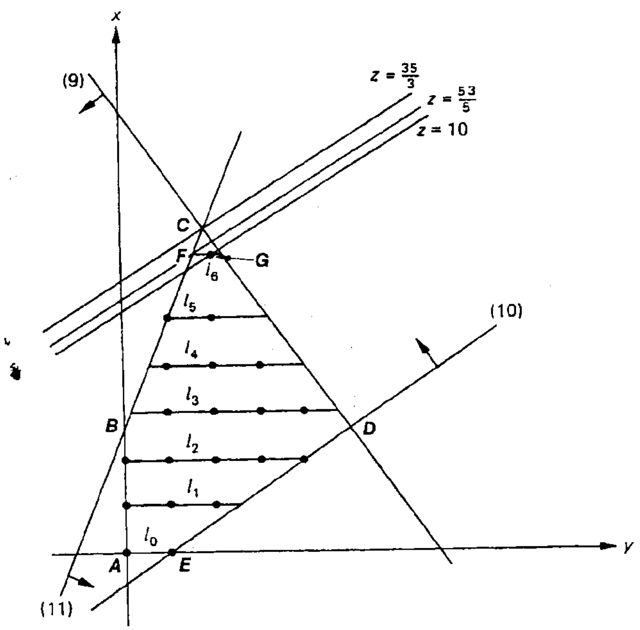


FIGURE 1.8

integer program occur at the intersection of the line segments l_i $(i=0,1,\ldots,6)$ with the hyperplanes $y=K$, where K is a nonnegative integer. The results are the 20 dotted points (Fig. 1.8). A parallel shift of the z hyperplane further into the polyhedron yields the optimal integer solution $\boldsymbol{G}=(2,6)$, with $z=10$. Note that $\boldsymbol{G}$ is an interior point.

Example 1.10 No Solution to the Integer Program

$$\begin{aligned}
&\text{Maximize} && x_1+x_2=z \\
&\text{subject to} && -4x_1+x_2\leqslant -1, && (13)\\
& && 4x_1+x_2\leqslant 3, && (14)\\
&\text{and} && x_1,\ x_2\geqslant 0 \text{ and integer.}
\end{aligned}$$

The linear programming feasible region is spanned by the extreme points $\boldsymbol{ABC}$ (Fig. 1.9) and the optimal solution to the LP occurs at $\boldsymbol{C}=(1/2,1)$. However, there are no integer points (x_1,x_2) inside the polyhedron; therefore the integer program has no feasible solution.

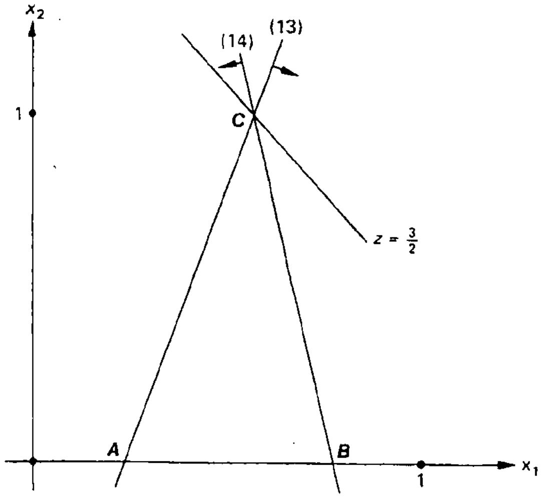


FIGURE 1.9

The previous discussion suggests that useful information can be extracted from the associated linear program; that is, the problem where the x integer requirements have been dropped. In particular, since the solutions to the MIP or IP are contained in the linear programming feasible region, we have

Theorem 1.2 The maximal value of the objective function to the mixed integer (or integer) program solved as a linear program is an upper bound on the value of any mixed integer (or integer) feasible solution.

Corollary 1.2.1 If the optimal solution of the mixed (or integer) problem solved as a linear one is integer in its integer constrained variables it solves the mixed (or integer) program.

Corollary 1.2.2 If the mixed (or integer) problem solved as a linear one is infeasible, then so is the mixed (or integer) program.

The geometry also suggests that we might solve the mixed (or integer) problem as a linear one and then round the integer constrained variables which are not integer either up or down. This procedure does in many instances produce a solution which may only slightly violate the constraints and is "good enough." This is especially true for problems where the accuracy of the data is questionable, or when the integer variables can take on large values. (For example, the effect of producing 732,333 parts rather than 732,334—or for that matter, 732,000—is negligible.) On the other hand, the optimal mixed (or integer) solution is often required when the integer variables take on smaller values, especially since the rounding may result in a point whose value is relatively far from an optimal solution, or it may violate crucial constraints. The next example demonstrates the shortcomings of rounding techniques. Other pitfalls of rounding are in Glover and Sommer [25] and in Problem 1.13.

Example 1.11 (Glover and Sommer [25])

Consider the following facility location problem (Example 1.5).

	Shipping costs g_{ij}					Production capacity
Customers	1	2	3	4	$5=n$	M_i
Sources						
1	93	70	48	68	81	2
2	45	89	97	85	96	3
3	92	93	58	37	99	2
4	55	103	55	57	38	3
$m=5$	74	60	78	54	52	2
Demands d_j	1	1	1	1	1	

Suppose that the fixed costs can be ignored (i.e., $f_i=0$ for $i=1,\ldots,5$) and the situation requires that if a facility is operating it must produce (and ship) its capacity. (In relation to the model, this means that the inequalities (3) in Example 1.5 become equalities.) As the total demand is 5, the latter assumption implies that there are just six combinations of sources which can supply the customers, namely (1,2), (1,4), (2,3), (2,5), (3,4), and (4,5). The problem with the corresponding pairs of variables (x_1,x_2), (x_1,x_4), etc. at 1 and the remaining x_i variables at 0 is a linear program. The optimal solutions to the six linear programs are listed below, where unlisted variables have value 0.

Facility variables	Shipping variables	Total cost
$x_1=x_2=1$	$z_{12}=z_{13}=z_{21}=z_{24}=z_{25}=1$	343
$x_1=x_4=1$	$z_{12}=z_{13}=z_{41}=z_{44}=z_{45}=1$	268
$x_2=x_3=1$	$z_{21}=z_{22}=z_{25}=z_{33}=z_{34}=1$	325
$x_3=x_4=1$	$z_{32}=z_{34}=z_{41}=z_{43}=z_{45}=1$	278
$x_2=x_5=1$	$z_{21}=z_{22}=z_{23}=z_{54}=z_{55}=1$	337
$x_4=x_5=1$	$z_{41}=z_{43}=z_{45}=z_{52}=z_{54}=1$	262 (optimal)

When the integer restrictions on the facility variables are dropped, the optimal linear programming solution is

$$x_1=x_3=x_5=1/2, \quad x_2=x_4=1/3, \quad z_{13}=z_{21}=z_{34}=z_{45}=z_{52}=1 \qquad (15)$$

with total cost 258. It is evident that this solution cannot be rounded to produce any of the six integer solutions since three of the x_i in (15) would have to be rounded down to 0. But then the solution becomes infeasible because the constraints

$$\sum_{j=1}^{5} z_{ij}=M_i x_i \qquad (i=1,\ldots,5)$$

are violated, since $x_i=0$ implies $z_{ij}=0$ for all j and in the solution (15) $z_{ij}=1$ for each i.

1.4 THE STATE OF THE ART

In 1949 Dantzig [11] showed that certain integer programs may be solved as linear ones.[5] (Well-known cases include the assignment, transportation, and static max flow problem.) Clearly however, most integer programs will not exhibit this integrality property, and the simplex method, per se, will generally not solve mixed integer (or integer) programs.

The principal approaches for solving such problems are (a) cutting plane techniques, (b) enumerative methods, (c) partitioning algorithms, and (d) group theoretic approaches. We briefly describe these while mentioning the principle contributors. The details of the algorithms are left to subsequent chapters.

1.4.1 Cutting Plane Techniques (Chapters 2, 3, 4, 5)

The general intent of cutting plane algorithms is to deduce supplementary inequalities from the integrality and constraint requirements which eventually produce a linear program whose optimal solution is integer in the integer constrained variables.

BACKGROUND. The constraint generation idea was proposed by Dantzig, Fulkerson, and Johnson [14] in 1954 in their work on the traveling salesman problem, and then in 1957 by Markowitz and Manne [49]. However, in 1958 Gomory [28] developed the first cutting plane algorithm applicable to any integer program. Shortly afterward, Gomory [30] and Beale [5] generalized Gomory's results to the mixed integer case. In 1960 Gomory [29] produced a second cutting plane algorithm for the integer program which requires only additions and subtractions in computation (an "all-integer" technique). All of the above methods maintain linear programs which are dual feasible, and are therefore often classified as dual cutting plane algorithms.

Glover [21] and Young [63] developed cutting plane algorithms for the integer program which maintain linear problems that are primal feasible. Since primal feasible integer solutions are successively produced, the technique is referred to as a primal cutting plane algorithm.

[5] Necessary and sufficient conditions for this to occur appear in Veinott and Dantzig [59]. Their results are repeated in Problem 1.14. Other related work is in Camion [8], Chandrasekaran [9], Glover [22], Heller [38], [39], [40], Heller and Hoffman [41], Hoffman and Kruskal [42], and in Thompson [57].

1.4.2 Enumerative Methods (Chapters 6, 7)

The intent here is to enumerate, either explicitly or implicitly, all possible solution candidates to the mixed integer or integer program. The feasible solution which maximizes the objective function is optimal. Enumerative algorithms can usually be characterized as a bookkeeping scheme which keeps track of the enumeration, coupled with a "point or node algorithm" whose objective is to show that certain related integer points cannot yield improved solutions; that is, via criteria which are contrived from the integrality and constraint requirements, to implicitly enumerate large numbers of points. The efficiency of the enumerative technique usually depends largely on the effectiveness of its point algorithm.

BACKGROUND. Land and Doig [45] in 1960 proposed an enumerative technique for the general mixed integer program. Little, Murty, and others [48] used the same idea in developing an algorithm for the traveling salesman problem. In 1965 Balas [2] proposed an enumeration scheme for solving the zero-one integer program. Extensions were developed by Glover [20]; Lemke and Spielberg [47], and Driebeek [15] expanded on the method, and presented an algorithm for the mixed problem. A survey of the earlier methods may be found in Lawler and Wood [46].

1.4.3 Partitioning Algorithms (Chapter 8)

Benders [6] in 1962 showed that a mixed integer program can be posed as an integer problem. The difficulty in solving the integer formulation arises out of the large number of constraints which must be generated. To overcome this obstacle Benders proposed a "partitioning algorithm" which solves the integer equivalent of the mixed problem by solving a series of related integer and linear programs.

1.4.4 Group Theoretic Algorithms (Chapter 9)

In the classical paper on a cutting plane algorithm, Gomory [28] discussed the relationships between the integer program and particular group structures.[6] Based on this work, Gomory [31] in 1965 showed that by relaxing the nonnegativity, but not integrality, constraints on certain variables, any integer program can be represented as a minimization problem defined on a group. Algorithms for solving this group problem and hence the integer program have appeared in the literature. The use of these problems in generating valid inequalities for any integer program, originally discussed by Gomory [32], has also been extensively treated; most recently by Gomory and Johnson [33], [34].

Although a very substantial effort has been (and is being) made to develop efficient algorithms for linear programs with discrete variables we cannot claim that every mixed integer or integer program can be solved in practice. There are known problems with relatively few integer variables that cannot be solved in a reasonable length of time using the existing techniques and computer facilities. The algorithms simply refuse to converge in some instances. It is not even clear, given a particular program, which method is best. Thus composite approaches are often tried.

Fortunately, most real world formulations yield highly structured constraint matrices. Often, special purpose algorithms that explicitly take advantage of the structure may be developed which are usually considerably more efficient than the classical techniques. These, however, do not preclude the use of study of the well-known methods. In fact, special purpose algorithms are usually variations of the classical techniques whose basic properties often support the finiteness of the specialized scheme.

At present, general algorithms (as listed in [54]) can usually solve problems with 50 to 100 variables and as many constraints. As the size of the problem grows beyond these limits, the probability of solving it in a reasonable amount of computer time decreases dramatically. On the other hand, special purpose techniques may be able to solve problems with a few thousand variables and several hundred rows (as in the case of set covering problems). The potential limits of the general and specialized algorithms are discussed in appendices devoted to computational experience which appear in later chapters. Other recent computational experience is discussed by Garfinkel and Nemhauser [18], and by Geoffrion and Marsten [19].

PROBLEMS

1.1 (Piecewise convex constraint sets) Suppose that some of the sets $\boldsymbol{b}_i - \boldsymbol{A}_i y \leqslant \mathbf{0}$ in Example 1.3 are unbounded above. Should the reformulation be modified? Illustrate your answer by solving

$$\begin{array}{lll} \text{maximize} & y_1 + y_2 & \\ \text{subject to} & y_1 \qquad \leqslant 2 & \text{and/or} \\ & \qquad y_2 \leqslant 3 & \\ \text{and} & y_1,\ y_2 \geqslant 0. & \end{array}$$

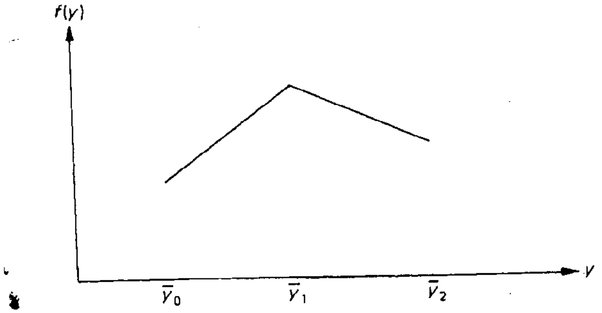
1.2 Suppose we formulate the piecewise convex constraint set problem appearing in Example 1.3 as the mixed integer *nonlinear* program

$$\begin{array}{lll} \text{maximize} & cy & \\ \text{subject to} & x_i(\boldsymbol{b}_i - \boldsymbol{A}_i y) \geqslant \mathbf{0} & (i = 1,\dots,p), \\ & \sum_{i=1}^{n} x_i = t, & \\ \text{and} & x_i = 0 \text{ or } 1 & (i = 1,\dots,p). \end{array}$$

Is this formulation correct? If so, discuss a solution technique which is computationally the same when applied to the mixed integer *linear* formulation appearing in Example 1.4. Illustrate.

[6] A group is a set which is closed under an (arithmetic) operation.

1.3 Consider the piecewise linear function $f(y)$ below, which is linear in two intervals.



Then an equivalent function (see Example 1.4) is:

$$f(y) = \alpha_0 f(\bar{y}_0) + \alpha_1 f(\bar{y}_1) + \alpha_2 f(\bar{y}_2),$$

where

$$y = \alpha_0 \bar{y}_0 + \alpha_1 \bar{y}_2 + \alpha_2 \bar{y}_3,$$

$$1 = \alpha_0 + \alpha_1 + \alpha_2,$$

$$x_0 \quad > \alpha_0,$$

$$x_0 + x_1 > \alpha_1,$$

$$x_1 > \alpha_2,$$

$$x_0 + x_1 = 1,$$

$$\alpha_0, \alpha_1, \alpha_2 > 0,$$

and

$$x_0, x_1 = 0 \text{ or } 1.$$

Are there redundant constraints on the x_i's and α_j's? Explain. Is your answer correct when there are n linear intervals?

1.4 (Raghavachari [51], [52], Kennington and Fyffe [44]) Consider the zero-one integer program

$$\begin{array}{llll} \text{(IP)} & \text{maximize} & cx, & \\ & \text{subject to} & Ax < b, & \\ & & 0 < x < e, & \\ & \text{and} & & x \text{ integer,} \end{array}$$

and the quadratic program

$$\begin{array}{lll} \text{(QP)} & \text{maximize} & cx - Mx^T(e-x), \\ & \text{subject to} & Ax < b, \\ & \text{and} & 0 < x < e, \end{array}$$

where e is a vector of ones, M is a large positive number, and x^T is the row vector x.

By proving the results below, show that the integer program may be solved by solving the quadratic program. Intuitively, why do these results seem true?

a) If x^* is an optimal solution to QP and x^* is integer, then it is optimal for IP.

b) If x^* is an optimal solution to QP and x^* does not have all integer components, then IP has no integer feasible solution.

c) If QP has no feasible solution then IP has no integer feasible solution.

1.5 Given the facility location problem with capacity constraints which appears in Example 1.5, show that the inequalities in the model imply that the shipping variables will be integer in an optimal solution. Is this result true for every optimal solution? Illustrate your answer using the data appearing in the next problem (part (d)) with the additional capacity restrictions $M_1 = 300$ and $M_2 = 350$. (This capacitated problem can be solved by inspection.)

1.6 (Facility location problem) Starting with the reformulated facility location problem appearing in Example 1.5:

a) Prove that there exists an optimal solution with all $y_{ij} = 0$ or 1.

b) Discuss when fractional solutions are possible.

c) Suggest what you consider to be an efficient solution procedure. Why does your scheme work?

d) Using the method obtained in (c), solve the two-plant, three-customer problem whose data appears below.

g_{ij} $i \backslash j$	1	2	3
1	5	7	6
2	5	6	7

d_j	1	2	3
	150	170	260

f_i	1	2
	90	100

1.7 (Plant location problem with side constraints) Remodel the plant location problem (Example 1.5) to allow for each of the following additional conditions. In each case can the model be simplified?

a) Only a subset of the plants may be open.

b) The cost of opening a plant is equal to a fixed cost K_i plus the amount produced at the plant.

c) The facilities can be listed according to regions, and because of customer convenience each region k must have at least l_k (a positive number) plants open.

d) A total amount of F dollars is available for construction.

1.8 (Airline crew scheduling with crew base constraints) Formulate the following problem as an integer (linear) program: An airline with n crews must make m flight legs. Each crew is currently in one of N locations (or cities) and the possible combination of flights each crew may take can be enumerated. Because of union rules, however, at most u_k and at least l_k crews ($k = 1,\ldots,N$) at city k may be assigned flight duty. The cost of using crew j on the ith combination of flights is c_{ij} ($j = 1,\ldots,n$). The problem is to obtain an assignment which covers all flights and satisfies the union rules at the minimum total cost.

1.9 (Multi-item knapsack problem) Consider the knapsack problem:

$$\begin{aligned} &\text{maximize} && \sum_{j=1}^{n} c_j x_j = z \\ &\text{subject to} && \sum_{j=1}^{n} a_j x_j < b, \\ &&& x_j > 0 \text{ and integer,} \end{aligned}$$

where c_j is the relative (positive) value of item j, a_j its (positive) weight, b is the (knapsack's) weight capacity, and x_j is the number of items j to put in the knapsack.

a) If x_j is not required to be integer, solve the eight variable problem whose data is

c_j	8	10	9	8	6	2	7	5
a_j	4	2	7	3	5	1	3	5

and $b=23$. (*Hint*: Let $z_j = a_j x_j$ for $j=1,\dots,n$ and transform the problem to the variables z_j. The linear programming solution is then obvious.)

b) Obtain an integer feasible solution by changing the value of one variable in the solution from (a).

c) Using (b), what is the range of values for an optimal integer solution? Why?

1.10 (Symmetric traveling salesman problem) Consider the traveling salesman problem with $d_{ij} = d_{ji}$ for all i,j. Can the integer programming model which appears in Example 1.8 be simplified?

1.11 (The four color problem) A classical unsolved problem in mathematics is the four color problem. The question is: Given any map which divides a plane into regions, can we use no more than four colors to color it so that no two regions with a common boundary have the same color.

a) Formulate the requirements as linear constraints with integer variables. (*Hint*: Let the variable in region i be denoted by x_i, where $x_i = 0,1,2,3$, depending on which color is used for that region.)

b) Why is it not possible to develop an objective function and solve the integer program (where the constraints are from (a)) so that the four color problem is answered?

1.12 Devise a two-variable integer program so that when it is solved as a linear program it produces a solution which, when "rounded" (i.e., when the fractional elements are set to the next largest or smallest integer), is "relatively far" (i.e., the objective function has substantially different values) from the integer optimal solution. Illustrate graphically.

1.13 (Glover and Sommer [25]) Consider the integer program

$$\begin{aligned} &\text{maximize} && cx \\ &\text{subject to} && Ax + Is = b, \\ &&& x > 0,\ s > 0 \text{ and integer,} \end{aligned}$$

where A is an m by n matrix of full rank (i.e., it has m linearly independent rows), I is an m identity matrix, and s are the slack variables. Suppose $x_B = B^{-1}b$ is an optimal basic feasible solution to the associated linear program (i.e., the integer restrictions are dropped), where B is a basis whose columns are from (A,I), and B^{-1} is the inverse of B.

a) From linear programming, we know that x_B is a unique vector. Does this mean that if it contains fractional values the solution can never be rounded to obtain an integer feasible solution? (*Hint*: x_B contains the basic variables; the nonbasic variables currently have value 0. Can some of them increase? Which ones?)

b) Show that whenever an integer programming problem does not contain slack variables, the optimal solution to the associated linear program can never be rounded to produce an integer feasible solution.

1.14 (Veinott and Dantzig [59]) Let A be a given integral matrix and let $X = \{x \mid Ax = b, x > 0\}$. Assume A has m linearly independent rows and thus a basis B is a subset of m linearly independent columns from A. We say that B is *unimodular* if the determinant B, written $|B|$, equals $+1$ or -1.

a) Prove that the following are equivalent.

i) Every basis is unimodular.

ii) The extreme points of X are integral for all integral b.

iii) Every basis has an integral inverse.

(*Hint*: Show (i)⇒(ii)⇒(iii)⇒(i). In general, use the fact that x_B is an extreme point if and only if $x_B = B^{-1}b$; appeal to $B^{-1} = B^{+}/|B|$, where B^{+} is the adjoint of B. To show (ii)⇒(iii), let y be any integral vector for which $y + B^{-1}e_i > 0$, where e_i is the ith identity column (it has a 1 in row i and zeroes everywhere else); let $z = y + B^{-1}e_i$ and demonstrate that z must be integral which will mean that the ith column of B^{-1} is integer.)

REFERENCES

1. Balas, E.; "Extension de l'algorithme additif á la programmation en nombres entriers et á la programmation non linéaire," *C. R. Acad. Sc. Paris*, May 1964.
2. ———; "An Additive Algorithm for Solving Linear Programs with Zero-One Variables", *Operations Research* 13(4), 517–548 (1965).
3. Balinski, M.; "Integer Programming: Methods, Uses, Computation," *Managemeni Science* 12(3), 253–313 (1965).
4. Balinski, M., and K. Spielberg; "Methods for Integer Programming: Algebraic, Combinatorial, and Enumerative," in *Progress in Operations Research* Vol. III (ed.: J. Arnofsky), 195–292, New York, John Wiley & Sons, 1968.
5. Beale, E.; "A Method of Solving Linear Programming Problems When Some But Not All of the Variables Must Take Integral Values," Statistical Techniques Research Group, Technical Report No. 19, Princeton University, July 1958.
6. Benders, J.; "Partitioning Procedures for Solving Mixed-Variables Programming Problems," *Numerische Mathematik* 4, 238–252 (1962).

7. Bowman, V., and F. Glover; "A Note on Zero-One Integer and Concave Programming," *Operations Research* **20**(1), 182–183 (1972).

8. Camion, P.; "Characterization of Totally Unimodular Matrices," *Proceedings of the American Mathematical Society* **16**, 1068–1073 (1965).

9. Chandrasekaran, R.; "Total Unimodularity of Matrices," *SIAM Journal of Applied Mathematics* **17**(6), 1032–1034 (1969).

10. Colville, A.; "Mathematical Programming Codes: Programs Available from IBM Program Information Department," IBM Philadelphia Scientific Center Report No. 320-2925, January 1968.

11. Dantzig, G.; "An Application of the Simplex Method to a Transportation Problem," in *Activity Analysis of Production and Allocation* (ed.: T. C. Koopmans), Cowles Commission Monograph No. 13, New York, John Wiley & Sons, 1951.

12. ———; "On the Significance of Solving Linear Programming Problems with Some Integer Variables," *Econometrica* **28**(1), 30–44 (1960).

13. ———; *Linear Programming and Extensions*, Princeton University Press, 1963.

14. Dantzig, G., D. Fulkerson, and S. Johnson; "Solution of a Large Scale Traveling Salesman Problem," *Operations Research* **2**(4), 393–410 (1954).

15. Driebeek, N.; "An Algorithm for the Solution of Mixed Integer Programming Problems," *Management Science* **12**(7), 576–587 (1966).

16. Faden, B. (ed.); *Computer Program Directory*, published by CCM Information Corporation, New York, for the Joint Users Group of the Association for Computing Machinery, 1971.

17. Fricks. R.; "Nonconvex Linear Programming," Ph.D. Thesis, Department of Operations Research, Case Western Reserve University, January 1971.

18. Garfinkel, R., and G. Nemhauser; "A Survey of Integer Programming Emphasizing Computation and Relation Among Models," Department of Operations Research Technical Report No. 156, Cornell University, August 1972.

19. Geoffrion, A., and R. Marsten; "Integer Programming: A Framework and State-of-the-Art Survey," *Management Science* **18**(9), 465–491 (1972).

20. Glover, F.; "A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem," *Operations Research* **13**(6), 879–929 (1965).

21. ———; "A New Foundation for a Simplified Primal Integer Programming Algorithm," *Operations Research* **16**(4), 727–748 (1968).

22. ———; "A Note on Linear Programming and Integer Feasibility," Management Sciences Research Report No. 127, Carnegie-Mellon University, August 1968.

23. ———; "Improved Linear Representations of Discrete Mathematical Programs," Management Science Report No. 72-8, the University of Colorado at Boulder, February 1972.

24. Glover, F., and D. Klingman; "Concave Programming Applied to a Special Class of Zero-One Integer Programs," Graduate School of Business Report No. AMM-11, the University of Texas at Austin, January 1969.

25. Glover, F., and D. Sommer; "Pitfalls of Rounding in Discrete Management Decision Problems," Management Science Report No. 72-2, the University of Colorado at Boulder, February 1972.

26. Glover, F., and R. Woolsey; "Further Reduction of Zero-One Polynomial Programming Problems to Zero-One Linear Programming Problems," *Operations Research* **21**(1), 156–161 (1973).

27. ———; "Converting the 0-1 Polynomial Programming Problem to a 0-1 Linear Program," *Operations Research*, **21**(1), 156–161 (1973).

28. Gomory, R.; "An Algorithm for Integer Solutions to Linear Programs," in *Recent Advances in Mathematical Programming* (eds.: Graves and Wolfe), New York, McGraw-Hill, 1963.

29. ———; "All Integer Programming Algorithm," IBM Research Center Report RC-189, January 1960; also in *Industrial Scheduling* (eds.: Muth and Thompson), Englewood Cliffs, NJ., Prentice-Hall, 1963.

30. ———; "An Algorithm for the Mixed Integer Problem," RM-2597 Rand Corporation, July 1960.

31. ———; "On the Relation Between Integer and Noninteger Solutions to Linear Programs," *Proceedings of the National Academy of Science* **53**(2), 260–265 (1965).

32. ———; "Faces of an Integer Polyhedron," *Proceedings of the National Academy of Science* **57**(1), 16–18 (1967).

33. Gomory, R., and E. Johnson; "Some Continuous Functions Related to Corner Polyhedra," *Mathematical Programming* **3**(1), 23–85 (1972).

34. ———; "Some Continuous Functions Related to Corner Polyhedra II,"*Mathematical Programming* **3**(3), 359–389 (1972).

35. Granot, F., and P. Hammer; "On the Role of Generalized Covering Problems," Center of Cybernetic Studies Research Report CS96, the University of Texas at Austin, September 1972.

36. Hadley, G.; *Linear Programming*, Reading, Mass., Addison-Wesley, 1962.

37. Hammer, P., and S. Rudeanu; *Boolean Methods in Operations Research and Related Areas*, New York, Springer-Verlag, 1968.

38. Heller, I.; "On Linear Systems with Integral Valued Solutions," *Pacific Journal of Mathematics* **7**, 1351–1364 (1957).

39. ———; "On a Class of Equivalent Systems of Linear Inequalities," *Pacific Journal of Mathematics* **13**, 1209–1227 (1963).

40. ———; "On Unimodular Sets of Vectors," in *Recent Advances in Mathematical Programming* (eds.: Graves and Wolfe), New York, McGraw-Hill, 1963.

41. Heller, I., and A. Hoffman; "On Unimodular Matrices,"*Pacific Journal of Mathematics* **12**, 1321–1327 (1962).

42. Hoffman, A., and J Kruskal; "Integral Boundary Points of Convex Polyhedra," in *Linear Inequalities and Related Systems* (eds.: Kuhn and Tucker), Princeton University Press, 1956.

43. Karp, R.; "Reducibility Among Combinatorial Problems," available from the University of California at Berkeley. (Paper presented at the Symposium on Mathematical Programming at the University of Wisconsin at Madison, September 1972.)

44. Kennington, J., and D. Fyffe; "A Note on the Quadratic Programming Approach to the Solution of the 0-1 Linear Integer Programming Problem," School of Industrial and Systems Engineering Report, Georgia Institute of Technology, 1972.

45. Land, A., and A. Doig; "An Automatic Method of Solving Discrete Programming Proglems," *Econometrica* **28**(3), 497–520 (1960).

46. Lawler, E., and D. Wood; "Branch and Bound Methods: A Survey," *Operations Research* **14**(4), 699–719 (1966).

47. Lemke, C., and K. Spielberg; "Direct Search Algorithm for Zero-One and Mixed Integer Programming," *Operations Research* **15**(5), 892–914 (1967).

48. Little, J., K. Murty, D. Sweeney, and C. Karel; "An Algorithm for the Traveling Salesman Problem," *Operations Research* **11**(5) 972–989 (1963).

49. Markowitz, H., and A. Manne; "On the Solution of Discrete Programming Problems," *Econometrica* **25**(1), 84–110 (1957).

50. Petersen, C.; "A Note on Transforming the Product of Variables to Linear Form in Linear Programs," Working Paper, Purdue University, 1971.

51. Raghavachari, M.; "On Connections Between Zero-One Integer Programming and Concave Programming Under Linear Constraints," *Operations Research* **17**(4), 680–684 (1969).

52. ———; "Supplement," *Operations Research* **18**(3), 564–565 (1970).

53. Rubin, D.; "Redundant Constraints and Extraneous Variables in Integer Programs," *Management Science* **18**(7), 423–427 (1972).

54. Salkin, H.; "A Brief Survey of Integer Programming," *OPSEARCH* **10**(2), 81–123 (1973).

55. Taha, H.; "A Balasian-Based Algorithm for 0-1 Polynomial Programming," Research Report No. 70-2, the University of Arkansas, May 1970.

56. Thompson, G., F. Tonge, and S. Zionts; "Techniques for Removing Constraints and Extraneous Variables from Linear Programming Problems," *Management Science* **12**(7), 588–608 (1966).

57. Thompson, R.; "Unimodular Group Matrices with Rational Integers as Elements," *Pacific Journal of Mathematics* **14**(2), 719–726 (1964).

58. Tucker, A.; "On Directed Graphs and Integer Programs," IBM Mathematical Research Project Technical Report, Princeton University, 1960.

59. Veinott, A., and G. Dantzig; "Integeral Extreme Points," *SIAM Review* **10**(3), 371–372 (1968).

60. Watters, L.; "Reduction of Integer Polynomial Programming Problems to Zero-One Linear Programming Problems," *Operations Research* **15**(6), 1171–1174 (1967).

61. White, W.; "On the Computational Status of Mathematical Programming," IBM Philadelphia Scientific Center Report No. 320-2990, June 1970.

62. Woolsey, R.; "A Candle to Saint Jude, or Four Real World Application of Integer Programming," available from the Colorado School of Mines, Golden, Colorado, 1971.

63. Young, R.; "A Simplified Primal (All-Integer) Integer Programming Algorithm," *Operations Research* **16**(4), 750–782 (1971).

64. Zangwill, W.; "Media Selection by Decision Programming," *Journal of Advertising Research* 5(3), (1965).

Some Applications-References

65. Ashour, S.; "An Experimental Investigation and Computational Evaluation of Flow Shop Scheduling Problems," *Operations Research* **18**(3), 541–548 (1970).

66. Balas, E.; "Machine Sequencing via Disjunctive Graphs: An Implicit Enumeration Algorithm," *Operations Research* **17**(6), 941–957 (1969).

67. Bowman, E.; "The Schedule-Sequencing Problem," *Operations Research* 7(5), 621–624 (1959).

68. Dantzig, G.; "A Machine-Job Scheduling Model," *Management Science* **6**(1), 191–196 (1960).

69. Freeman, R., D. Gogerty, G. Graves, and R. Brooks; "A Mathematical Model of Supply Support for Space Operations," *Operations Research* **14**(1), 1–15 (1966).

70. Giglio, R., and H. Wagner; "Approximate Solutions to the Three-Machine-Scheduling Problem," *Operations Research* **12**(2), 306–324 (1964).

71. Gupta, J.; "The Generalized *n*-Job, *M*-Machine Scheduling Problem," *OPSEARCH* **8**(3), 171–185 (1971).

72. Kirby, M., and P. Scobey; "Production Scheduling of *N* Identical Machines," *Canadian Operational Research* **8**(1), 14–27 (1970).

73. Lind, M.; "Application of Mixed Integer Programming to Letter Processing Systems," Mitre Corporation Report M71-102, October 1971.

74. Manne, A.; "On the Job-Shop Scheduling Problem," *Operations Research* **8**(2), 219–223 (1960).

75. Merville, L.; "An Investment Decision Model for the Multinational Firm: A Chance-Constrained Programming Approach," Ph.D. Thesis, University of Texas, May 1971.

76. Moder, J., and C. Phillips; *Project Management with CPM and PERT*, New York, Reinhold, 1964.

77. Story, A., and H. Wagner; "Computational Experience with Integer Programming for Job Shop Scheduling," in *Industrial Scheduling* (eds.: Muth and Thompson), Englewood Cliffs, N.J., Prentice-Hall, 1963.

78. Von Lanzenauer, C., and R. Himes; "A Linear Programming Solution to the General Sequencing Problem," *Canadian Operational Research* **8**(2), 129–134 (1970).

79. Wagner, H.; "An Integer Linear Programming Model for Machine Scheduling," *Naval Research Logistics Quarterly* **6**, 131–140 (1959).

80. Wagner, H., R. Giglio, and R. Glaser; "Preventive Maintenance Scheduling by Mathematical Programming," *Management Science* **10**(2), 316–334 (1964).

EXHIBIT F

5.1 Smoothing techniques for continuous responses

In this section a short survey of smoothing techniques for the continuous case is given. Observations are bivariate data $(y_i, x_i), i = 1, \ldots, n$, where the response y_i and the explanatory variable x_i are measured on interval scale level. It is assumed that the dependence of y_i on x_i is given by

$$y_i = f(x_i) + \varepsilon_i$$

where f is an unspecified smooth function and ε_i is a noise variable with $E(\varepsilon_i) = 0$. A scatterplot smoother $\hat{f}$ is a smooth estimate of f based on observations (y_i, x_i), $i = 1, \ldots, n$. Most of the smoothers considered in the following are linear. That means if one is only interested in the fit at an observed value x_i the estimate has the linear form

$$\hat{f}(x_i) = \sum_{j=1}^{n} s_{ij} y_j \, .$$

The weights $s_{ij} = s(x_i, x_j)$ depend on the target point x_i where the response is to be estimated and on the value x_j where the response y_j is observed.

5.1.1 Simple neighbourhood smoothers

A simple device for the estimation of $f(x_i)$ is to use the average of response values in the neighbourhood of x_i. These *local average estimates* have the form

$$\hat{f}(x_i) = \mathrm{Ave}_{j \in N(x_i)}(y_j)$$

where Ave is an averaging operator and $N(x_i)$ is a neighbourhood of x_i. The extension of the neighbourhood is determined by the *span* or *window size* w, which denotes the proportion of total points in a neighbourhood.

Let the window size w be from (0,1) and let the integer part of wn, denoted by $[wn]$, be odd. Then a symmetric neighbourhood may be constructed by

$$N(x_i) = \{\max\{i - \frac{[wn]-1}{2}, 1\}, \ldots, i-1, i, i+1, \ldots \min\{i + \frac{[wn]-1}{2}, n\}\}.$$

$N(x_i)$ gives the indices of the ordered data $x_1 < \ldots < x_n$. It contains x_i and $([wn]-1)/2$ points on either side of x_i if x_i is from the middle of the data. The neighbourhood is smaller near the end points, e.g., at x_1, x_n, which leads to quite biased estimates at the boundary. Of course w determines the smoothness of the estimate. If w is small the estimate is very rough; for large w the estimate is a smooth function.

A special case of a local average estimate is the *running mean* where Ave stands for arithmetic mean. Alternatively the mean may be replaced by the *median*, yielding an estimate that is more resistant to outliers. A drawback of the median is that the resulting smoother is nonlinear.

Another simple smoother is the *running–line smoother*. Instead of computing the mean a linear term

$$\hat{f}(x_i) = \hat{\alpha}_i + \hat{\beta}_i x_i$$

is fitted where $\hat{\alpha}_i, \hat{\beta}_i$ are the least–squares estimates for the data points in the neighbourhood $N(x_i)$. In comparison to the running mean, the running–line smoother reduces bias near the end points. However, both smoothers, running mean and running line may produce curves that are quite jagged.

If the target point x is not from the sample $x_1, \ldots, x_n$ one may interpolate linearly between the fit of the two $\hat{y}_i$ values, which are observed at predictor values from the sample adjacent to x. Alternatively, one may use varying nonsymmetric neighbourhoods. For $k \in \mathbb{N}$ the *k–nearest neighbourhood* (k–NN) estimate is a weighted average based on the varying neighbourhood

$$N(x) = \{i | x_i \text{ is one of the } k\text{–nearest observations to } x\}$$

where near is defined by a distance measure $d(x, x_i)$. Then the degree of smoothing is determined by k. The proportion of points in each neighbourhood is given by $w = k/n$. For example, the linear k–NN estimate has the form

$$\hat{f}(x) = \sum_{i=1}^{n} s(x, x_i) y_i$$

with weights

$$s(x, x_i) = \begin{cases} 1/k, & x_i \in N(x) \\ 0, & \text{otherwise.} \end{cases}$$

The weights fulfill the condition $\Sigma_i s(x, x_i) = 1$. The estimate is not restricted to unidimensional x–values. By appropriate choice of a distance measure, e.g., the Euclidian distance $d(x, x_i) = \|x_i - x\|^2$, the method may be applied to vector–valued x.

5.1.2 Spline smoothing

Cubic smoothing splines

Smoothed estimators may be considered compromises between faith with the data and reduced roughness caused by the noise in the data. This view